Profitina

Design and Analysis of Algorithms

The theoretical backbone of the whole curriculum: how to model a computational problem precisely, design an algorithm for it, prove that algorithm correct, and reason formally about how its running time grows — before a single line of production code gets written.

Irfan Subakti & Rully Soelaiman

profitina.com

Design and Analysis of Algorithms

From First Principles to Proven Programs

Irfan Subakti & Rully Soelaiman

Profitina

Copyright © 2026 Irfan Subakti & Rully Soelaiman. All rights reserved.

Published by Profitina · profitina.com

First edition, September 2026.

Every program in this book was compiled and run before its output was printed. The complete source code is in the accompanying Code folder. The programs are offered for teaching and learning; Irfan Subakti & Rully Soelaiman and the publisher accept no liability for their use elsewhere.

Contents

Preface ..... **lxviii**
Who this book is for lxviii
What is in it lxviii
How to read it lxix
About the authors lxix
Acknowledgments lxix

CHAPTER 1 Foundations, Framework, and First Problems **1**
1.1 Where We Begin 1
1.2 Algorithms and Data Structures — the Two Pillars 2
1.2.1 What an algorithm is 2
1.2.2 A program is an implementation of an algorithm 3
1.2.3 What a data structure is 3
1.2.4 The goals we are aiming at 4
1.3 The Two Goals of Algorithm Design 4
1.3.1 Goal number one: correctness 4
1.3.2 A problem stated precisely: sorting 5
1.3.3 Insertion sort, and the loop invariant that proves it 6
1.3.4 Goal number two: efficiency 9
1.3.5 The rule of thumb that decides everything 9
1.3.6 The two goals work together 10
1.4 The Five Design Paradigms 10
1.4.1 Incremental, or brute force 10
1.4.2 Greedy 11
1.4.3 Divide and conquer 11
1.4.4 Dynamic programming 12
1.4.5 Mathematical and number-theoretic 13
1.4.6 The five paradigms in one table 15
1.5 The Five-Phase Problem-Solving Framework 15
1.5.1 Phase 1 — Understand 15
1.5.2 Phase 2 — Abstract 16
1.5.3 Phase 3 — Design 16
1.5.4 Phase 4 — Analyze 16
1.5.5 Phase 5 — Implement 16
1.6 Problem 1 — Columns (E-Olymp 10281) 17
1.6.1 Phase 1 — Understand 17
1.6.2 Phase 2 — Abstract 18
1.6.3 Phase 3 — Design 20
1.6.4 Phase 4 — Analyze 20
1.6.5 Phase 5 — Implement 21
1.7 Problem 2 — Journey (E-Olymp 10280) 21
1.7.1 Phase 1 — Understand 21
1.7.2 Phase 2 — Abstract 22
1.7.3 Phase 3 — Design 23
1.7.4 Phase 4 — Analyze 23
1.7.5 Phase 5 — Implement 24
1.8 Problem 3 — Unique Numbers (SPOJ DCEPC901) 24
1.8.1 Phase 1 — Understand 24
1.8.2 Phase 2 — Abstract 25
1.8.3 Phase 3 — Design 27

1.8.4 Phase 4 — Analyze ... 27
1.8.5 Phase 5 — Implement ... 27
1.9 Problem 4 — Minimum and Maximum Numbers (SPOJ NEWLANG) ... 28
1.9.1 Phase 1 — Understand ... 29
1.9.2 Phase 2 — Abstract ... 30
1.9.3 Phases 3 and 4 — Design and Analyze ... 32
1.9.4 Phase 5 — Implement ... 32
1.10 Problem 5 — Interesting Sequence (E-Olymp 4894) ... 33
1.10.1 Phase 1 — Understand ... 33
1.10.2 Phase 2 — Abstract ... 34
1.10.3 Phases 3 and 4 — Design and Analyze ... 36
1.10.4 Phase 5 — Implement ... 36
1.11 Problem 6 — Recursive Function 1 (E-Olymp 10296) ... 37
1.11.1 Phase 1 — Understand ... 38
1.11.2 Phase 2 — Abstract ... 38
1.11.3 Phases 3 and 4 — Design and Analyze ... 41
1.11.4 Phase 5 — Implement ... 41
1.12 All Six Problems at a Glance ... 42
1.12.1 Recognizing which paradigm applies ... 43
1.13 Where These Problems Come From: the Online Judges ... 43
1.14 Judge Problems as Research: Two Papers from ITS ... 44
1.14.1 Extrapolating a periodic function ... 45
1.14.2 Reading handwritten digits ... 47
1.15 The Books, the Road Map, and the Evaluation Chapters ... 49
1.15.1 The books ... 49
1.15.2 The road map ... 50
1.15.3 The evaluation chapters ... 51
1.16 Setting Up Your Workbench: VS Code and a C++ Compiler ... 51
1.16.1 Windows, step by step ... 52
1.16.2 macOS, step by step ... 52
1.16.3 Linux, step by step ... 53
1.16.4 VS Code on every system ... 53
1.16.5 Your first program, compiled and run ... 53
1.16.6 Feeding a judge problem its input ... 54
1.16.7 Debugging: watching the program think ... 54
1.17 Profitina in Practice ... 54
1.17.1 The worked case study: Profitina Laundry ... 55
1.18 Common Mistakes, and How to Avoid Them ... 56
1.19 Chapter Summary ... 57
1.20 Exercises ... 58
1.21 Solutions to the Exercises ... 58
References ... 60

CHAPTER 2 Getting Started: the RAM Model, Insertion Sort, and Searching ... 62
2.1 Where We Left Off ... 62
2.2 What This Book Is Really About ... 63
2.2.1 What this book IS about ... 63
2.2.2 What this book is NOT about ... 63
2.3 A Brief History of Algorithms ... 64
2.3.1 Where the word comes from ... 64
2.3.2 The first algorithm ... 64
2.3.3 The nineteenth century: machines and the first programmer ... 65
2.3.4 The twentieth century: what computation *is* ... 66
2.4 The Vocabulary: Problem, Instance, Algorithm, Program, Data Structure ... 67
2.4.1 An algorithmic problem ... 67

2.4.2 An instance of a problem ... 68
2.4.3 An algorithm ... 68
2.4.4 A program ... 69
2.4.5 A data structure ... 69
2.4.6 Correctness and efficiency, defined ... 69
2.5 The Five Design Goals ... 70
2.6 The RAM Model of Computation ... 71
2.6.1 The rules of the model ... 71
2.6.2 What counts as one step ... 71
2.6.3 What does NOT count as one step ... 72
2.6.4 Why choose this model? ... 72
2.7 How to Analyze Running Time ... 73
2.7.1 What is the input size? ... 73
2.7.2 Running time is not wall-clock time ... 73
2.7.3 A first complete example ... 74
2.8 Insertion Sort: the Algorithm ... 74
2.8.1 The intuition: sorting a hand of cards ... 75
2.8.2 The algorithm in pseudocode ... 76
2.8.3 Tracing the algorithm by hand ... 76
2.8.4 Insertion Sort in C++ ... 77
2.9 Proving Insertion Sort Correct: the Loop Invariant ... 79
2.9.1 What a loop invariant is ... 79
2.9.2 The invariant for Insertion Sort ... 80
2.9.3 Initialization ... 80
2.9.4 Maintenance ... 81
2.9.5 Termination ... 82
2.10 Counting the Cost of Insertion Sort ... 82
2.10.1 The cost-and-times table ... 82
2.10.2 The general formula ... 83
2.11 Best Case, Worst Case, and Average Case ... 84
2.11.1 The best case: an array already sorted ... 84
2.11.2 The worst case: an array in reverse order ... 85
2.11.3 The average case ... 87
2.11.4 Checking the analysis against a machine ... 88
2.11.5 Why we focus on the worst case ... 91
2.12 The Growth of Functions ... 91
2.12.1 Logarithms, in words ... 92
2.12.2 The comparison table ... 92
2.12.3 Seeing it as a picture ... 93
2.12.4 The hierarchy of growth rates ... 95
2.12.5 A practical rule of thumb ... 95
2.13 Searching: Linear Search ... 96
2.13.1 The problem ... 96
2.13.2 The algorithm ... 96
2.13.3 A loop invariant for linear search ... 97
2.13.4 Analysis ... 97
2.13.5 Linear search in C++ ... 98
2.13.6 Can we do better? ... 98
2.14 Binary Search ... 98
2.14.1 The problem ... 98
2.14.2 The key idea ... 99
2.14.3 The algorithm ... 99
2.14.4 Tracing binary search by hand ... 100
2.14.5 Why it is $O(\lg n)$ ... 101
2.14.6 Binary search in C++, and the overflow trap ... 101

2.14.7 Binary search as recursion 103
2.14.8 The trade-off: preprocessing 105
2.15 Design Paradigms Revisited 105
2.15.1 The incremental approach 105
2.15.2 Divide and conquer 105
2.15.3 Ad hoc algorithms 106
2.16 Connecting Chapter 1 and Chapter 2 106
2.16.1 Where this chapter sits in the book 107
2.17 Profitina in Practice 108
2.17.1 The runnable case study: Profitina Laundry 109
2.18 Common Mistakes, and How to Avoid Them 112
2.19 Chapter Summary 113
2.20 Exercises 114
2.21 Solutions to the Exercises 115
Further reading 121
References 121

CHAPTER 3 Asymptotic Notation, Recurrences, and the Master Theorem 122
3.1 Where We Left Off 122
3.1.1 Partial correctness and total correctness 123
3.1.2 What is still missing 123
3.2 Why We Need a Language for Growth 124
3.2.1 The numbers, so you believe it 124
3.3 Big-O: The Asymptotic Upper Bound 126
3.3.1 A proof, done completely: $7n - 3 = O(n)$ 127
3.3.2 A proof where the threshold does real work: $3n2 + 5n + 7 = O(n2)$ 128
3.3.3 A disproof: $n2 \neq O(n)$ 128
3.3.4 Letting the machine hunt for witnesses 128
3.4 Big-Ω: The Asymptotic Lower Bound 130
3.5 Big-Θ: The Asymptotically Tight Bound 131
3.6 Little-o, Little-ω, and the Number Analogy 132
3.6.1 The number analogy 133
3.6.2 An abuse of notation we all commit 133
3.7 Properties of the Notations 133
3.7.1 Transitivity 133
3.7.2 Reflexivity 134
3.7.3 Symmetry and transpose symmetry 134
3.7.4 Sums, products, and constants 134
3.7.5 Why we drop constants and lower-order terms — proved, not asserted 135
3.7.6 Where the number analogy breaks: trichotomy fails 135
3.8 Standard Notations and Common Functions 136
3.8.1 Monotonicity 136
3.8.2 Floors and ceilings 136
3.8.3 Modular arithmetic 136
3.8.4 Polynomials 136
3.8.5 Exponentials 136
3.8.6 Logarithms — and the notation we will use everywhere 137
3.8.7 Factorials and Stirling's approximation 138
3.8.8 The iterated logarithm 138
3.8.9 Fibonacci numbers 139
3.9 Summations: the Arithmetic of Loops 141
3.9.1 The arithmetic series 141
3.9.2 The same result by induction 143
3.9.3 The geometric series 144
3.9.4 The harmonic series 145

3.9.5 Telescoping sums 145
3.9.6 Bounding a sum by an integral 146
3.10 Recurrences: Equations That Describe Recursion 149
3.10.1 What a recurrence is 149
3.10.2 The general divide-and-conquer form 150
3.10.3 Three technicalities we are allowed to ignore 150
3.10.4 The three methods 150
3.11 Method 1: The Substitution Method 151
3.11.1 A complete example: $T(n) = 4T(n/2) + n$ is $O(n3)$ 151
3.11.2 The failed attempt: is it $O(n2)$? 152
3.11.3 The fix: strengthen the hypothesis 152
3.11.4 Two more substitution techniques 153
3.11.5 Checking the algebra against the actual numbers 154
3.12 Method 2: Iteration and the Recursion Tree 156
3.12.1 Pure iteration: $T(n) = n + 3T(\lfloor n/4 \rfloor)$ 156
3.12.2 Drawing the tree: $T(n) = T(n/4) + T(n/2) + n2$ 157
3.12.3 When every level costs the same: $T(n) = T(n/3) + T(2n/3) + n$ 158
3.12.4 Counting the leaves: where $n\log ba$ comes from 159
3.12.5 Three shapes of tree 160
3.13 Method 3: The Master Theorem 162
3.13.1 The statement 162
3.13.2 What the pieces mean, in words 163
3.13.3 The role of ε: why "polynomially" 163
3.13.4 The regularity condition, in plain language 163
3.13.5 The summary card 164
3.13.6 Where the theorem refuses to answer: the gaps 164
3.13.7 The five-step strategy 165
3.14 Twenty Master Theorem Problems, Worked in Full 166
3.14.1 Binary search: $T(n) = T(n/2) + 1$ 166
3.14.2 $T(n) = 9T(n/3) + n$ 166
3.14.3 Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$ 166
3.14.4 $T(n) = 3T(n/4) + n\lg n$ 167
3.14.5 $T(n) = 4T(n/2) + n3$ 167
3.14.6 The gap: $T(n) = 2T(n/2) + n\lg n$ 167
3.14.7 $T(n) = 4T(n/2) + n$ 168
3.14.8 $T(n) = 4T(n/2) + n2$ 168
3.14.9 $T(n) = 8T(n/2) + n2$ 168
3.14.10 Strassen: $T(n) = 7T(n/2) + n2$ 168
3.14.11 $T(n) = 8T(n/2) + n3$ 169
3.14.12 $T(n) = 2T(n/4) + n$ 169
3.14.13 $T(n) = 3T(n/3) + n/2$ 169
3.14.14 $T(n) = 16T(n/4) + n2$ 169
3.14.15 $T(n) = T(n/2) + n$ 169
3.14.16 The other gap: $T(n) = 2T(n/2) + n/\lg n$ 170
3.14.17 $T(n) = 5T(n/2) + n2$ 170
3.14.18 $T(n) = T(2n/3) + 1$ 170
3.14.19 Regularity earning its keep: $T(n) = 2T(n/2) + n2/\lg n$ 170
3.14.20 Regularity failing: $T(n) = T(n/2) + n(2 - \cos n)$ 171
3.14.21 Checking all of them by machine 171
3.15 Merge Sort, Derived from Nothing 175
3.15.1 The divide-and-conquer blueprint 175
3.15.2 The algorithm 176
3.15.3 A complete trace on $A = [1,5,2,4,6,3,2,6]$ 176
3.15.4 Proving MERGE correct with a loop invariant 176
3.15.5 The recurrence, and three ways to solve it 177

3.15.6 The program 178
3.15.7 What Merge Sort costs, and what it buys 180
3.16 Profitina in Practice 180
3.16.1 The worked case study: Profitina Laundry 181
3.17 Common Mistakes, and How to Avoid Them 184
3.18 Chapter Summary 185
3.18.1 Looking ahead 186
3.19 Exercises 186
3.20 Solutions to the Exercises 187
Further reading 189
References 189

CHAPTER 4 Evaluation and Discussion I: Foundations, Analysis, and First Problems 190
4.1 Where We Stand 190
4.2 How to Read an Algorithms Problem 191
4.2.1 Step 1 — Copy the input and output formats, verbatim 191
4.2.2 Step 2 — Name the unknown, and name the data 192
4.2.3 Step 3 — Turn the constraints into a time budget 192
4.2.4 Step 4 — Compute the first five cases by hand 192
4.2.5 Step 5 — Name the shape 193
4.2.6 Step 6 — Write the invariant, the recurrence, or the formula, and prove it 193
4.2.7 Step 7 — Only now, write the program 193
4.3 Problem 1 — LDCSUMAB: Sums of Two Powers of Two 194
4.3.1 The problem in our own words 194
4.3.2 Steps 1 to 3: the formats, the symbols, the budget 195
4.3.3 Step 4: the first cases by hand 195
4.3.4 Steps 5 and 6: the shape, and the formula 196
4.3.5 Proof of correctness 198
4.3.6 Complexity 199
4.3.7 Step 7: the program 200
4.3.8 An alternative: binary search on the block 202
4.3.9 Checking against brute force 203
4.3.10 Common mistakes on this problem 204
4.4 Problem 2 — TOHU: Help Tohu 205
4.4.1 The problem in our own words 205
4.4.2 Steps 1 to 3: the formats, the symbols, the budget 205
4.4.3 Step 4: the first cases by hand 206
4.4.4 Steps 5 and 6: recovering ak by differencing 206
4.4.5 Summing by telescoping 207
4.4.6 Proof of correctness 209
4.4.7 Complexity, precision and overflow 210
4.4.8 Step 7: the program 210
4.4.9 An alternative: exact fractions, with no formula assumed 212
4.4.10 Common mistakes on this problem 213
4.5 Worked Problems on Correctness: Invariants and Traces 213
4.5.1 Worked Problem A — a loop invariant for Horner's rule 214
4.5.2 Worked Problem B — tracing Insertion Sort 217
4.5.3 Worked Problem C — tracing binary search 219
4.6 Worked Problems on Counting and Growth 222
4.6.1 Worked Problem D — the exact step count of a triple loop 222
4.6.2 Worked Problem E — asymptotic claims with explicit constants 224
4.6.3 Worked Problem F — ranking thirteen functions by growth 227
4.7 Worked Problems on Recurrences 230
4.7.1 Worked Problem G — the substitution method, and a hypothesis that must be strengthened 230

4.7.2 Worked Problem H — a recursion tree with a table of levels 232
4.7.3 Worked Problem I — four recurrences and the Master Theorem, including a gap 235
4.8 Worked Problems on Closed Forms and Hidden Recurrences 238
4.8.1 Worked Problem J — Laundry Tokens 238
4.8.2 Worked Problem K — tiling a $2 \times N$ strip 240
4.9 From Brute Force to Better: a Timed Comparison 243
4.9.1 Worked Problem L — does any pair sum to x? 243
4.10 The Problems of This Chapter Compared 246
4.11 Profitina in Practice 248
4.11.1 The worked case study: Profitina Laundry 248
4.12 Common Mistakes, and How to Avoid Them 252
4.13 Chapter Summary 253
4.14 Exercises 254
4.15 Solutions to the Exercises 255
References 257

CHAPTER 5 Sorting in Practice: Heapsort and Quicksort 258
5.1 Where We Left Off 258
5.1.1 The toolkit so far 258
5.1.2 The frustrating gap 259
5.1.3 Why $\Theta(n\lg n)$ is the target, and what sorting costs in a real system 259
5.2 The Selection Sort Insight 260
5.3 The Binary Heap: a Tree Hidden Inside an Array 261
5.3.1 Nearly complete binary trees 261
5.3.2 The index arithmetic, derived 261
5.3.3 The heap property 263
5.3.4 A concrete heap 263
5.3.5 A heap is not sorted 264
5.3.6 The height of a heap is $\lfloor \lg n \rfloor$ 265
5.3.7 Where the leaves live 265
5.3.8 How many nodes sit at each height 266
5.3.9 All of it, in a program 266
5.4 `MAX-HEAPIFY`: the Engine That Keeps the Heap Honest 268
5.4.1 The problem it solves 268
5.4.2 The algorithm 269
5.4.3 A complete walkthrough 269
5.4.4 Why it is correct 272
5.4.5 The running time 273
5.4.6 `MAX-HEAPIFY` in C++ 274
5.5 `BUILD-MAX-HEAP`: the Surprising Linear-Time Construction 276
5.5.1 The algorithm 276
5.5.2 Correctness, by loop invariant 277
5.5.3 A complete walkthrough 277
5.5.4 The loose bound: $O(n\lg n)$ 281
5.5.5 The tight bound: $O(n)$, derived step by step 281
5.5.6 The identity $h = 0\infty h/2h = 2$, proved 282
5.5.7 Seeing the linear cost on a real machine 284
5.6 Heapsort 286
5.6.1 The algorithm 286
5.6.2 A complete trace 286
5.6.3 Correctness, by loop invariant 288
5.6.4 The running time 289
5.6.5 Why Heapsort is not stable 290
5.6.6 Heapsort in C++ 290
5.7 Priority Queues 292

5.7.1 What a priority queue is 292
5.7.2 Where priority queues are used 292
5.7.3 `MAXIMUM`: the easy one 293
5.7.4 `EXTRACT-MAX` 293
5.7.5 `INCREASE-KEY` 293
5.7.6 `INSERT` 294
5.7.7 What the four operations cost 295
5.7.8 A complete priority queue in C++ 295
5.7.9 Using the standard library instead 298
5.8 Quicksort 300
5.8.1 The idea 300
5.8.2 Divide and conquer, with the work moved 300
5.8.3 Lomuto's `PARTITION`, and the four-region invariant 301
5.8.4 A Lomuto partition, traced by hand 302
5.8.5 Hoare's `PARTITION`, the original scheme 302
5.8.6 Hoare's partition, traced twice 304
5.8.7 The Quicksort procedure itself 307
5.8.8 Quicksort, traced completely 308
5.9 Analyzing Quicksort: Best, Worst, and Average 310
5.9.1 Everything depends on the split 310
5.9.2 The best case: perfectly balanced splits 310
5.9.3 The worst case: maximally unbalanced splits 311
5.9.4 The balanced-split intuition, and the 9: 1 split 313
5.9.5 The average case, argued two ways 314
5.9.6 Checking the theory against a machine 318
5.10 Randomized Quicksort 320
5.10.1 The fix 320
5.10.2 What randomization actually buys 320
5.10.3 Other pivot strategies 321
5.10.4 Tail recursion and the stack 321
5.11 The $\Omega(n\lg n)$ Lower Bound for Comparison Sorts 324
5.11.1 What counts as a comparison sort 324
5.11.2 The decision tree 325
5.11.3 The proof 325
5.11.4 What the bound does and does not forbid 326
5.12 Choosing a Sort: the Honest Comparison 327
5.12.1 The complete landscape 327
5.12.2 Stability, and why anyone cares 328
5.12.3 Cache behavior, and why Quicksort wins in practice 330
5.12.4 When to use what 334
5.13 Profitina in Practice 335
5.13.1 Where this chapter appears in a real product 335
5.13.2 The worked case study: Profitina Laundry 336
5.14 Common Mistakes, and How to Avoid Them 340
5.15 Chapter Summary 341
5.15.1 Looking ahead 342
5.16 Exercises 343
5.17 Solutions to the Exercises 344
References 349

CHAPTER 6 Binary Search Trees and AVL Trees 350
6.1 Where We Left Off 350
6.2 The Vocabulary of Trees 351
6.3 What a Binary Search Tree Is 352
6.4 The BST Property Is Not the Heap Property 353

6.5 Representing a Binary Search Tree in C++ 354
6.6 Walking the Tree: Three Traversals 355
6.6.1 Why the inorder walk is sorted — a proof 359
6.7 Searching 359
6.8 Minimum and Maximum 361
6.9 Successor and Predecessor 361
6.10 Insertion 365
6.11 Deletion: the Three Cases 366
6.11.1 Case 1 — the node is a leaf 366
6.11.2 Case 2 — the node has exactly one child 367
6.11.3 Case 3 — the node has two children 368
6.12 A Complete Binary Search Tree in C++ 369
6.13 When the Tree Goes Bad: h Can Reach n 373
6.14 AVL Trees: Balance as an Invariant 376
6.15 Why an AVL Tree Is Short: the Minimum-Nodes Recurrence 378
6.15.1 The Fibonacci connection 379
6.15.2 From Fibonacci to $1.44\lg n$ 380
6.16 Rotations: the One Tool 382
6.16.1 Right rotation, with the exact pointer updates 383
6.16.2 Left rotation 385
6.16.3 What a rotation does to the height 386
6.17 The Four Rotation Cases 386
6.17.1 Case LL — one right rotation 387
6.17.2 Case LR — a left rotation then a right rotation 388
6.17.3 Case RR — one left rotation 389
6.17.4 Case RL — a right rotation then a left rotation 390
6.17.5 Telling the four cases apart in code 390
6.18 A Worked Insertion Example, Step by Step 391
6.18.1 Step 1 — insert 14: no rebalancing needed 391
6.18.2 Step 2 — insert 3: the Left-Left case 392
6.18.3 Step 3 — insert 45: the Right-Right case 393
6.18.4 Step 4 — insert 7: the Left-Right case 394
6.18.5 Step 5 — insert 15: the Right-Left case 396
6.19 A Complete AVL Tree in C++ 397
6.19.1 A reference implementation in Java 401
6.20 AVL Deletion 403
6.20.1 Why y and x are chosen differently 404
6.20.2 Why insertion needs at most one rebalancing, and deletion may need $\lg n$ of them 404
6.20.3 A deletion example, step by step 408
6.21 Choosing: AVL Tree, Red-Black Tree, or Hash Table 410
6.21.1 Red-black trees 410
6.21.2 Hash tables 410
6.22 Profitina in Practice 414
6.22.1 The worked case study: Profitina Laundry 415
6.23 Common Mistakes, and How to Avoid Them 419
6.24 Chapter Summary 422
6.24.1 Looking ahead 423
6.25 Exercises 423
6.26 Solutions to the Exercises 424
References 433

CHAPTER 7 From Divide and Conquer to Dynamic Programming 434
7.1 Where We Left Off 434
7.1.1 What is still missing 435
7.2 Divide and Conquer as a Design Discipline 435

7.2.1 The three steps 435
7.2.2 Merge Sort, seen once more as a shape 436
7.2.3 When divide and conquer is the right tool 436
7.2.4 When divide and conquer wastes work 437
7.3 Fibonacci: the Smallest Problem That Breaks Divide and Conquer 437
7.3.1 The definition 437
7.3.2 The straightforward recursion 438
7.3.3 The recursion tree 438
7.3.4 Counting the calls exactly 439
7.3.5 Why the cost is $\Theta(\phi n)$ 439
7.3.6 How much work is actually repeated 441
7.3.7 Measuring it 441
7.4 Four Repairs to Fibonacci 443
7.4.1 Repair 1: memoization — remember what you have already answered 444
7.4.2 Repair 2: bottom-up — build the small answers first 444
7.4.3 Repair 3: $\Theta(1)$ space 445
7.4.4 Repair 4 (bonus): the matrix power, $O(\lg n)$ 446
7.4.5 All four, measured 446
7.5 Optimization Problems 450
7.5.1 What makes a problem an optimization problem 450
7.5.2 Optimal substructure, stated precisely 450
7.5.3 The cut-and-paste argument 451
7.5.4 A problem *without* optimal substructure 451
7.6 Multiplying a Chain of Matrices 452
7.6.1 The cost of one matrix product 452
7.6.2 Associativity, and why the grouping matters 452
7.6.3 A four-matrix example, priced 453
7.6.4 How many parenthesizations are there? 454
7.6.5 The subproblem, and the recurrence derived 456
7.6.6 Counting the subproblems 458
7.6.7 The algorithm 459
7.6.8 The table filled by hand, cell by cell 460
7.6.9 The c table and the reconstruction 461
7.6.10 $\Theta(n3)$ time and $\Theta(n2)$ space 463
7.6.11 The complete program 464
7.7 Memoization: Keeping the Recursion 468
7.7.1 Lookup-Chain 468
7.7.2 Measured: plain recursion, memoized, bottom-up 469
7.7.3 The trade-off, stated 472
7.8 The Four Steps of Designing a Dynamic-Programming Algorithm 472
7.8.1 Step 1: characterize the structure of an optimal solution 472
7.8.2 Step 2: recursively define the value of an optimal solution 473
7.8.3 Steps 3 and 4: compute the value, then construct the solution 473
7.8.4 The recipe, on one page 474
7.9 Longest Common Subsequence 475
7.9.1 Why anyone cares 475
7.9.2 Subsequence, defined 475
7.9.3 Brute force, and why it fails 476
7.9.4 Optimal substructure, proved 476
7.9.5 The recurrence 477
7.9.6 The algorithm 478
7.9.7 The table filled by hand 479
7.9.8 The arrow table 481
7.9.9 Reconstructing the LCS 481
7.9.10 $\Theta(mn)$, and the $\Theta(\min(m,n))$-space variant 482

7.9.11 The complete program 483
7.10 Overlapping Subproblems versus Independent Subproblems 486
7.10.1 The two hallmarks 486
7.10.2 Measured 487
7.10.3 The decision procedure 489
7.11 Profitina in Practice 490
7.11.1 The worked case study: Profitina Laundry 491
7.12 Common Mistakes, and How to Avoid Them 494
7.13 Chapter Summary 495
7.13.1 Looking ahead 496
7.14 Exercises 497
7.15 Solutions to the Exercises 498
References 506

CHAPTER 8 Evaluation and Discussion II: Sorting, Search Trees, and Dynamic Programming 507
8.1 Where We Stand 507
8.2 How to Approach a Multi-Part Problem 509
8.3 Problem 8.1 — Building a Max-Heap, Step by Step 510
8.3.1 The problem 510
8.3.2 What you need from Chapter 5 511
8.3.3 Step 1: the array as a tree 511
8.3.4 Step 2: the trace, one call at a time 512
8.3.5 Step 3: heapsort on the result 513
8.3.6 Step 4: why the construction costs only $O(n)$ 514
8.3.7 The program 514
8.3.8 Check yourself 517
8.3.9 Common wrong turns on this problem 518
8.4 Problem 8.2 — Quicksort's Partition, Traced; Best and Worst Cases 518
8.4.1 The problem 518
8.4.2 What you need from Chapter 5 518
8.4.3 Step 1: the partition, one step per line 519
8.4.4 Step 2: the whole recursion 520
8.4.5 Step 3: the worst case 521
8.4.6 Step 4: the best case 521
8.4.7 The program 522
8.4.8 Check yourself 526
8.4.9 Common wrong turns on this problem 527
8.5 The Data Type the Tree Problems Use 527
8.6 Problem 8.3 — `isBST`: Is This Tree a Binary Search Tree? 529
8.6.1 The problem 529
8.6.2 What the problem is really about 530
8.6.3 The wrong answer, and the smallest tree that exposes it 530
8.6.4 Deriving the right answer 531
8.6.5 The program 532
8.6.6 Why this is not the heap property 535
8.6.7 A tempting but wrong solution, and why it fails 535
8.6.8 The alternative method: the inorder walk 536
8.6.9 Common wrong turns on this problem 536
8.7 Problem 8.4 — `printDescending`: Largest First, With No List 537
8.7.1 The problem 537
8.7.2 What the problem is really about 537
8.7.3 Deriving the answer 538
8.7.4 The program 538
8.7.5 The alternative method: an explicit stack 541

8.7.6 Common wrong turns on this problem 541
8.8 Problem 8.5 — `max`: the Largest Key Along One Path 541
8.8.1 The problem 541
8.8.2 What the problem is really about 542
8.8.3 Deriving the answer 542
8.8.4 The program 542
8.8.5 $O(h)$, $O(\lg n)$, and how to state the bound so that it is true 545
8.8.6 The $\leq$ in the hint 545
8.8.7 The empty tree 546
8.8.8 Common wrong turns on this problem 546
8.9 Problem 8.6 — `isHeightBalanced`: the AVL Condition in One Pass 546
8.9.1 The problem 546
8.9.2 What the AVL condition is 547
8.9.3 The obvious answer, and what it costs 548
8.9.4 The answer: one pass, with a sentinel 549
8.9.5 The program 550
8.9.6 The recurrence, solved with the Master Theorem 554
8.9.7 Common wrong turns on this problem 555
8.10 Problem 8.7 — `insertHB`: Insertion That Keeps the Tree Balanced 555
8.10.1 The problem 555
8.10.2 The contract, stated precisely 555
8.10.3 Deriving `insertHB` 556
8.10.4 A first look at the machine in action 557
8.11 Problem 8.8 — `rebalanceForLeft`: the Cases LL and LR 557
8.11.1 The problem 557
8.11.2 The one tool: rotation 557
8.11.3 Case LL — one right rotation 558
8.11.4 Case LR — a left rotation, then a right rotation 559
8.11.5 The subtlest point: why $\leq$ and not $<$ 560
8.12 Problem 8.9 — `rebalanceForRight`: the Cases RR and RL 561
8.12.1 The problem 561
8.12.2 The mirror principle 561
8.12.3 Cases RR and RL, in pictures 561
8.12.4 Common wrong turns on Problems 8.8 and 8.9 562
8.13 Problem 8.10 — `deleteHB`: Deletion That Keeps the Tree Balanced 563
8.13.1 The problem 563
8.13.2 The three deletion cases 563
8.13.3 The trap: which rebalancer to call 564
8.13.4 Why deletion may need many rebalancings, and insertion at most one 565
8.13.5 Predecessor or successor? 565
8.13.6 Problems 8.7 to 8.10 as one program 567
8.13.7 Common wrong turns on this problem 572
8.14 Problems 8.11 to 8.15 — Red-Black Trees 572
8.14.1 The problems, and how they relate to the AVL problems 572
8.14.2 The five red-black properties 573
8.14.3 Problem 8.11 — counting the black nodes 574
8.14.4 Problem 8.12 — no red node with a red child 574
8.14.5 Problem 8.13 — a uniform black-height 574
8.14.6 Problem 8.14 — the complete validator 574
8.14.7 Problem 8.15 — rotations with parent pointers 574
8.14.8 The five problems as one running program 575
8.15 Problem 8.16 — Matrix-Chain Multiplication, the Table Filled by Hand 582
8.15.1 The problem 582
8.15.2 What you need from Chapter 7 582
8.15.3 Step 1: the dimensions and the diagonal 583

8.15.4 Step 2: chains of length 2 583
8.15.5 Step 3: chains of length 3 583
8.15.6 Step 4: chains of length 4 583
8.15.7 Step 5: the whole chain 584
8.15.8 Step 6: reading the bracketing, and checking it 584
8.15.9 Step 7: plain recursion against memoization 585
8.15.10 The program 585
8.15.11 Check yourself 590
8.15.12 Common wrong turns on this problem 590
8.16 Problem 8.17 — Longest Common Subsequence, the Table and the Walk Back 590
8.16.1 The problem 590
8.16.2 What you need from Chapter 7 590
8.16.3 Step 1: the table, one cell per line 591
8.16.4 Step 2: the walk back 593
8.16.5 Step 3: is SORE the only answer? 593
8.16.6 Step 4: the cost 593
8.16.7 The program 594
8.16.8 Check yourself 597
8.16.9 Common wrong turns on this problem 597
8.17 Twenty-Five Quick Checks 597
8.18 Further Worked Problems: an Order-Statistic Tree 598
8.18.1 The problems 599
8.18.2 How each solution is derived 599
8.18.3 Complete solutions 600
8.18.4 Reading the output as a check 606
8.19 Profitina in Practice 607
8.19.1 The worked case study: Profitina Laundry 608
8.20 Common Mistakes, and How to Avoid Them 614
8.21 Chapter Summary 615
8.21.1 Looking ahead 616
8.22 Exercises 616
8.23 Solutions to the Exercises 617
References 626

CHAPTER 9 Greedy Algorithms 627
9.1 Where We Left Off 627
9.1.1 What if the choice were obvious? 627
9.2 What a Greedy Algorithm Is 628
9.2.1 The definition 628
9.2.2 The uncomfortable fact 628
9.2.3 What the rest of the chapter does 629
9.3 A Motivating Failure: Making Change 629
9.3.1 The problem 629
9.3.2 The greedy rule 630
9.3.3 The counterexample 630
9.3.4 A coin system where greed succeeds 631
9.3.5 Measured 631
9.4 The Activity-Selection Problem 635
9.4.1 The problem 635
9.4.2 The running example 636
9.4.3 Four candidate greedy rules 636
9.4.4 Optimal substructure, proved 638
9.4.5 The dynamic program we are about to throw away 640
9.4.6 The greedy choice, and why only one subproblem remains 640
9.4.7 The greedy-choice property, proved by an exchange argument 642

9.4.8 Putting the two halves together 643
9.4.9 The recursive greedy algorithm 643
9.4.10 The trace on the running example 644
9.4.11 The iterative greedy algorithm 646
9.4.12 The running time 648
9.4.13 The complete program 648
9.5 The Elements of the Greedy Strategy 653
9.5.1 The six-step procedure 653
9.5.2 Ingredient one: the greedy-choice property 654
9.5.3 Ingredient two: optimal substructure 654
9.5.4 How to recognize the ingredients: a practical recipe 655
9.6 Greedy versus Dynamic Programming: the Two Knapsacks 655
9.6.1 The two problems 655
9.6.2 The greedy rule, and the instance 656
9.6.3 Fractional: greedy succeeds 656
9.6.4 0-1: greedy fails 658
9.6.5 How badly, and how often? 658
9.6.6 The two techniques, side by side 659
9.6.7 The complete program 660
9.7 Huffman Codes 664
9.7.1 The problem 664
9.7.2 The running example, with the bit counts computed 664
9.7.3 Prefix-free codes, and why they matter 666
9.7.4 Codes as binary trees 666
9.7.5 The cost of a tree 667
9.7.6 The algorithm 668
9.7.7 The trace, step by step, on the running example 669
9.7.8 Correctness, part one: the greedy-choice lemma 672
9.7.9 Correctness, part two: the optimal-substructure lemma 673
9.7.10 The theorem 674
9.7.11 The running time 674
9.7.12 The complete encoder and decoder 675
9.8 Matroids and Greedy Methods 681
9.8.1 The question this section answers 681
9.8.2 The definition, and what every word means 681
9.8.3 A worked example: the graphic matroid 682
9.8.4 Two consequences of the exchange property 683
9.8.5 Weighted matroids and the greedy algorithm 684
9.8.6 The application: scheduling unit-time tasks with deadlines and penalties 686
9.8.7 Worked by hand on an example 687
9.8.8 One algorithm, two problems 689
9.9 A Catalog of Greedy Algorithms Still to Come 694
9.10 Profitina in Practice 695
9.10.1 The worked case study: Profitina Laundry 695
9.11 Common Mistakes, and How to Avoid Them 702
9.12 Chapter Summary 703
9.12.1 Looking ahead 705
9.13 Exercises 705
9.14 Solutions to the Exercises 707
References 718

CHAPTER 10 Elementary Graph Algorithms: Breadth-First and Depth-First Search 719
10.1 Where We Left Off 719
10.1.1 What is new, and what carries over 720
10.2 What a Graph Is 720

10.2.1 Start with something you can see 720
10.2.2 Writing it down 721
10.2.3 The formal definition 721
10.2.4 Two small conventions 722
10.3 What Graphs Are For 722
10.4 The Vocabulary of Graphs 724
10.4.1 Adjacent, incident, neighbor 724
10.4.2 Degree 725
10.4.3 The handshake lemma, proved 725
10.4.4 In-degree and out-degree 726
10.4.5 Walks, paths, and simple paths 727
10.4.6 Cycles and acyclic graphs 729
10.4.7 Connected graphs 730
10.4.8 Connected components 730
10.4.9 Strongly connected components, for directed graphs 731
10.4.10 Subgraphs and spanning subgraphs 731
10.4.11 Trees and forests 732
10.4.12 Complete graphs 733
10.4.13 Bipartite graphs 733
10.4.14 Weighted graphs 734
10.4.15 How many edges can there be? Sparse and dense 734
10.4.16 The whole vocabulary on one page 735
10.5 Three Ways to Store a Graph 736
10.5.1 What a representation has to do 736
10.5.2 The edge list 737
10.5.3 The adjacency list 739
10.5.4 The adjacency matrix 740
10.5.5 Space, side by side 741
10.5.6 The four questions, side by side 742
10.5.7 Choosing 742
10.5.8 All three, in one program 742
10.6 Searching a Graph 746
10.7 Breadth-First Search 747
10.7.1 The labyrinth and the string 747
10.7.2 The three things BFS computes 748
10.7.3 The algorithm 748
10.7.4 A worked example, one step at a time 750
10.7.5 Reading the answer off 753
10.7.6 The program 754
10.7.7 Why it costs $O(V + E)$ 758
10.7.8 The breadth-first tree 759
10.7.9 BFS computes shortest distances: the proof 760
10.8 Depth-First Search 763
10.8.1 The labyrinth, the string, and the can of paint 763
10.8.2 The algorithm 764
10.8.3 The two timestamps 766
10.8.4 A worked example, one step at a time 766
10.8.5 The depth-first forest 769
10.8.6 Without recursion 769
10.8.7 Why it costs $\Theta(V + E)$ 774
10.8.8 The parenthesis theorem 775
10.8.9 The white-path theorem 779
10.8.10 Classifying the edges 780
10.9 Six Things the Two Searches Can Do 782
10.9.1 Connected components 782

10.9.2 Cycle detection 784
10.9.3 Testing bipartiteness 787
10.9.4 Shortest paths in an unweighted graph 790
10.9.5 Directed acyclic graphs 791
10.9.6 Topological sort 792
10.9.7 So: breadth-first or depth-first? 797
10.9.8 Measured 797
10.10 Profitina in Practice 800
10.10.1 The worked case study: Profitina Laundry 801
10.11 Common Mistakes, and How to Avoid Them 808
10.12 Chapter Summary 809
10.12.1 Looking ahead 810
10.13 Exercises 810
10.14 Solutions to the Exercises 812
References 818

CHAPTER 11 Minimum Spanning Trees, DAGs, and Topological Sort 820
11.1 Where We Left Off 820
11.1.1 What is still missing 821
11.2 Directed Acyclic Graphs 821
11.2.1 Cycles, and what it means to have none 821
11.2.2 Four everyday DAGs 822
11.2.3 The DAG theorem 823
11.2.4 Proving the DAG theorem 823
11.2.5 Why the "first vertex" choice matters 824
11.3 Topological Sort 825
11.3.1 What we want 825
11.3.2 Why a topological order is (almost) never unique 825
11.3.3 The running example: getting dressed 826
11.3.4 Algorithm 1: depth-first search, and the finishing times 827
11.3.5 The example traced, one timestamp at a time 827
11.3.6 Why it is correct 829
11.3.7 The running time is $\Theta(V + E)$ 829
11.3.8 The complete program 830
11.3.9 Algorithm 2: Kahn's in-degree algorithm 832
11.3.10 Kahn traced on the same DAG 832
11.3.11 Why Kahn is correct 833
11.3.12 Kahn's running time 834
11.3.13 The complete program, with cycle detection 834
11.3.14 Which of the two to use 837
11.4 Spanning Trees 837
11.4.1 Definition 838
11.4.2 A spanning tree has exactly $V - 1$ edges 838
11.4.3 How many spanning trees are there? 839
11.5 The Minimum Spanning Tree Problem 839
11.5.1 Weights 839
11.5.2 A real problem with costs 840
11.5.3 An example graph 840
11.5.4 Why brute force is hopeless, and what we need instead 841
11.6 Optimal Substructure of the MST 841
11.6.1 The claim 841
11.6.2 The proof 842
11.6.3 What optimal substructure alone does not give us 843
11.7 The Cut Property 843
11.7.1 Cuts, crossing edges, and respecting a set 843

11.7.2 The theorem 844
11.7.3 The proof, in a picture 844
11.7.4 Reading it once more, slowly 845
11.7.5 The corollary the algorithms actually use 846
11.7.6 A worked check 846
11.8 The Generic MST Algorithm 846
11.8.1 Safe edges 846
11.8.2 The generic algorithm 847
11.8.3 The loop invariant 847
11.8.4 The more specific version 847
11.8.5 Two ways to choose the cut 848
11.9 Prim–Jarník 848
11.9.1 The idea: one growing cloud 848
11.9.2 The two arrays: key and π 848
11.9.3 The algorithm 849
11.9.4 Priority queues 849
11.9.5 The example traced, step by step 849
11.9.6 Why Prim is correct 854
11.9.7 The running time 854
11.9.8 Three queues compared 854
11.9.9 The complete program 855
11.9.10 A note on the lazy `DecreaseKey` 859
11.10 Kruskal 859
11.10.1 The idea: a forest that merges 859
11.10.2 Disjoint sets, from scratch 860
11.10.3 Attempt 1: sets as lists 860
11.10.4 Attempt 2: sets as trees 861
11.10.5 Trick 1: union by rank 861
11.10.6 Trick 2: path compression 862
11.10.7 Why the amortized cost is almost constant 862
11.10.8 The disjoint-set program 863
11.10.9 The algorithm 866
11.10.10 The example traced, edge by edge 866
11.10.11 Why Kruskal is correct 872
11.10.12 The running time 872
11.10.13 The complete program 873
11.11 Prim versus Kruskal 875
11.11.1 The two answers, compared 875
11.11.2 Sparse and dense 875
11.11.3 Measured 876
11.11.4 The decision table 879
11.12 Equal Weights, and When the MST Is Unique 879
11.12.1 Ties break the algorithms apart, not the answer 879
11.12.2 Distinct weights imply a unique MST 880
11.12.3 Testing uniqueness 880
11.13 Profitina in Practice 881
11.13.1 The scan pipeline is a DAG, and the scheduler topologically sorts it 881
11.13.2 Minimum spanning trees in near-duplicate detection 882
11.13.3 The worked case study: Profitina Laundry 882
11.14 Common Mistakes, and How to Avoid Them 886
11.15 Chapter Summary 888
11.15.1 Looking ahead 888
11.16 Exercises 889
11.17 Solutions to the Exercises 890
References 895

CHAPTER 12 Evaluation and Discussion III: Greedy Algorithms and Graphs 896
12.1 Where We Stand 896
12.1.1 What Chapters 9, 10 and 11 built 896
12.1.2 A revision map 897
12.1.3 How this chapter is organized 899
12.2 A Design Project, Worked End to End 899
12.2.1 What a design project is, and its nine stages 899
12.2.2 Choosing a problem that deserves an algorithm 900
12.2.3 Stage 1: the problem, stated 901
12.2.4 Stage 2: the model 901
12.2.5 Stage 3: the algorithms chosen, and the alternatives rejected 902
12.2.6 Stage 4: the pseudocode 904
12.2.7 Stage 5: the complexity, worked out in advance 904
12.2.8 Stage 6: the design proved by hand, before any code 905
12.2.9 Stage 7: the implementation 907
12.2.10 Stage 8: the evaluation 914
12.2.11 Stage 9: the conclusion 917
12.2.12 Five pitfalls of a design project, and a check yourself 918
12.3 Further Worked Problems I: Greedy Algorithms 918
12.3.1 How the worked problems are laid out 918
12.3.2 Worked Problem A — making change: a greedy rule disproved, another proved 920
12.3.3 Worked Problem B — activity selection, run and proved 924
12.3.4 Worked Problem C — minimizing maximum lateness 927
12.3.5 Worked Problem D — a Huffman code, built step by step 929
12.3.6 Worked Problem E — the two knapsacks 933
12.3.7 Worked Problem F — a matroid at work: unit-time tasks with deadlines 935
12.3.8 Problems B to E as one program 940
12.4 Further Worked Problems II: Searching a Graph 947
12.4.1 The two graphs of this section 947
12.4.2 Worked Problem G — breadth-first search, traced 948
12.4.3 Worked Problem H — depth-first search of a digraph, every arc classified 950
12.4.4 Worked Problem I — depth-first search of an undirected graph 954
12.4.5 Worked Problem J — connected components 959
12.4.6 Worked Problem K — bipartiteness, and its certificate 960
12.4.7 Worked Problem L — cycle detection, undirected and directed 961
12.4.8 Problems G, H, J, K and L as one program 962
12.5 Further Worked Problems III: DAGs and Topological Sort 969
12.5.1 The prerequisite DAG of this section 969
12.5.2 Worked Problem M — is it a DAG? 969
12.5.3 Worked Problem N — topological sort by finishing time 970
12.5.4 Worked Problem O — topological sort by Kahn's algorithm 971
12.5.5 Worked Problem P — counting the topological orders 973
12.5.6 Worked Problem Q — the critical path 974
12.5.7 Problems M to Q as one program 975
12.6 Further Worked Problems IV: Minimum Spanning Trees 981
12.6.1 The weighted graph of this section 981
12.6.2 Worked Problem R — Kruskal, with a disjoint-set trace 981
12.6.3 Worked Problem S — Prim, with the key array 983
12.6.4 Worked Problem T — Prim, with a binary heap 984
12.6.5 Worked Problem U — the cut property and the cycle property 990
12.6.6 Worked Problem V — union by rank and path compression 991
12.6.7 Worked Problem W — is the minimum spanning tree unique? 993
12.6.8 Problems R, S, U, V and W as one program 994
12.7 Quick Self-Check Drills 1001
12.8 Profitina in Practice 1003

12.8.1 The worked case study: Profitina Laundry 1004
12.9 Common Mistakes, and How to Avoid Them 1009
12.10 Chapter Summary 1010
12.10.1 Looking ahead 1010
12.11 Exercises 1011
12.12 Solutions to the Exercises 1012
12.12.1 The programs 1014
12.12.2 Where to read further 1020
References 1021

CHAPTER 13 Single-Source Shortest Paths 1022
13.1 Where We Left Off 1022
13.1.1 What carries over from Chapters 10 and 11 1022
13.1.2 What is new 1023
13.2 The Shortest-Path Problem, Stated Precisely 1023
13.2.1 From "number of edges" to "total weight" 1023
13.2.2 The weight of a path 1023
13.2.3 The shortest-path weight 1024
13.2.4 A worked example, by hand 1024
13.2.5 The four variants of the problem 1025
13.2.6 Why single-source is the one we solve 1025
13.2.7 Where the problem actually turns up 1026
13.3 Optimal Substructure of Shortest Paths 1027
13.3.1 The theorem 1027
13.3.2 The proof, by cutting and pasting 1027
13.3.3 Reading the proof again, slowly 1028
13.3.4 What optimal substructure buys us 1028
13.4 The Triangle Inequality 1028
13.4.1 Statement 1028
13.4.2 The proof 1028
13.4.3 The edge form, which is the one we use 1029
13.4.4 A worked check 1030
13.5 Negative Weights and Negative Cycles 1030
13.5.1 A negative edge is not a problem 1030
13.5.2 A negative cycle is fatal 1030
13.5.3 Shortest paths have no cycles 1031
13.5.4 Which algorithm survives what 1032
13.6 Relaxation: the One Primitive 1032
13.6.1 The two arrays: d and π 1032
13.6.2 Initialization 1033
13.6.3 The relaxation step 1033
13.6.4 A worked relaxation 1033
13.6.5 The upper-bound property 1034
13.6.6 The no-path property 1035
13.6.7 The convergence property 1035
13.6.8 The path-relaxation property 1036
13.6.9 The predecessor-subgraph property 1036
13.6.10 The five properties on one page 1037
13.7 Dijkstra's Algorithm 1037
13.7.1 The idea 1037
13.7.2 The pseudocode 1038
13.7.3 Why the weights must be non-negative 1039
13.7.4 The example, one step at a time 1041
13.7.5 The same trace as a table 1044
13.7.6 The program 1045

13.7.7 What it prints 1048
13.7.8 Correctness 1048
13.7.9 Running time 1051
13.7.10 Measured 1052
13.7.11 Dijkstra, Prim, and breadth-first search 1055
13.8 The Bellman–Ford Algorithm 1055
13.8.1 Why we need it 1055
13.8.2 The algorithm 1055
13.8.3 Why $V - 1$ passes are enough 1056
13.8.4 The example, pass by pass 1058
13.8.5 The same trace as a table 1061
13.8.6 The program 1061
13.8.7 What it prints 1065
13.8.8 Negative-cycle detection, and why the extra pass finds it 1065
13.8.9 Running time 1066
13.9 Shortest Paths in a Directed Acyclic Graph 1067
13.9.1 Why a DAG is easy 1067
13.9.2 The algorithm 1067
13.9.3 Why no priority queue is needed 1068
13.9.4 A worked example 1068
13.9.5 The program 1069
13.9.6 Running time 1072
13.9.7 Longest paths and the critical path 1072
13.10 Reconstructing the Path, Not Just Its Length 1072
13.10.1 The predecessor array is the answer 1073
13.10.2 The program 1073
13.10.3 Three traps 1076
13.11 Which Algorithm for Which Graph 1076
13.12 Profitina in Practice 1077
13.12.1 Where shortest paths live inside an AI-visibility platform 1077
13.12.2 The worked case study: Profitina Laundry 1077
13.13 Common Mistakes, and How to Avoid Them 1081
13.14 Chapter Summary 1082
13.14.1 Looking ahead 1083
13.15 Exercises 1083
13.16 Solutions to the Exercises 1084
References 1088

CHAPTER 14 Amortized Analysis 1089
14.1 Where We Left Off 1089
14.1.1 A debt from Chapter 11 1089
14.1.2 What you need to have with you 1089
14.2 The Question Amortized Analysis Answers 1090
14.2.1 A bound that is true, and useless 1090
14.2.2 The factor of n, made concrete 1091
14.2.3 Three costs, and the only inequality that matters 1091
14.2.4 Amortized analysis is not average-case analysis 1092
14.2.5 The three methods 1093
14.3 The Aggregate Method 1094
14.3.1 What the method is 1094
14.3.2 The stack with `Multipop` 1094
The naive bound, once more 1095
The observation that fixes it 1096
The total, computed 1096
Measuring it 1097

14.3.3 The k-bit binary counter 1099
The 8-bit table 1100
How often does bit i flip? 1100
Adding it up 1101
Measuring it 1102
14.3.4 Exercise 17.1-3, worked in full 1105
Understanding the cost function 1106
Splitting the sum 1106
Evaluating it, one line at a time 1107
Turning it into a bound 1107
Checking it against a program 1108
14.3.5 What the aggregate method cannot do 1110
14.4 Amortized Is Not Average-Case — Said Again 1110
14.5 The Accounting Method 1111
14.5.1 Charges and credit 1111
14.5.2 The invariant, and why it is what you must prove 1112
14.5.3 The stack, by the accounting method 1112
14.5.4 The binary counter, by the accounting method 1113
Watching the credit 1114
14.5.5 Exercise 17.2-2, worked in full 1117
The charge 1118
The credit after i operations 1118
Case 1: i is exactly a power of two, $i = 2k$ with $k \geq 0$ 1118
Case 2: i lies strictly between powers, $i = 2k + j$ with $k \geq 0$ and $0 \leq j < 2k$ 1119
The conclusion 1119
Checked against a program 1119
14.6 The Potential Method 1121
14.6.1 The idea 1121
14.6.2 The telescoping sum 1122
14.6.3 The one condition, and why it is needed 1123
14.6.4 The stack, by the potential method 1124
14.6.5 The binary counter, by the potential method 1125
Both potentials, watched 1126
14.6.6 Exercise 17.3-6: a queue from two stacks 1128
The problem 1128
The design 1129
Why it is correct 1129
Why the naive cost bound is bad 1129
The potential function 1130
The program 1131
14.7 Dynamic Tables 1134
14.7.1 The problem 1134
14.7.2 `TABLE-INSERT` 1135
14.7.3 The aggregate method applied to `TABLE-INSERT` 1136
14.7.4 The accounting method applied to `TABLE-INSERT` 1137
14.7.5 The potential method applied to `TABLE-INSERT` 1138
The picture 1139
The program 1140
14.7.6 Deletion, and the temptation to halve at half 1143
The thrashing sequence, written out 1143
The fix 1144
Measuring both policies 1144
14.7.7 The potential function for the combined case 1147
14.8 Union–Find: The Promise from Chapter 11, Kept 1148
14.8.1 What we are analyzing 1148

14.8.2 Trick 1: union by rank 1149
14.8.3 Trick 2: path compression 1149
14.8.4 What the answer is 1149
14.8.5 What α actually is 1150
14.8.6 Why $\alpha(n) \leq 4$ for any n that fits in the universe 1151
14.8.7 Measuring it 1151
14.9 Choosing a Method 1155
14.10 Profitina in Practice 1156
14.10.1 Where amortization lives inside an AI-visibility platform 1156
14.10.2 The worked case study: Profitina Laundry 1157
14.11 Common Mistakes, and How to Avoid Them 1161
14.12 Chapter Summary 1162
14.12.1 Looking ahead 1163
14.13 Exercises 1163
14.14 Solutions to the Exercises 1164
References 1181

CHAPTER 15 All-Pairs Shortest Paths 1182
15.1 Where We Left Off 1182
15.1.1 What Chapter 13 gave us 1182
15.1.2 The new question 1183
15.2 The All-Pairs Problem, Precisely 1184
15.2.1 The input: an adjacency matrix 1184
15.2.2 The output: two matrices, not one 1186
15.2.3 Printing a path out of Π 1187
15.2.4 A first sanity check on sizes 1188
15.3 The Obvious Algorithm, and Its Price 1188
15.3.1 All weights non-negative: $V \times$ Dijkstra 1188
15.3.2 Negative edges present: $V \times$ Bellman–Ford 1189
15.3.3 Why we cannot simply use Dijkstra anyway 1189
15.3.4 What we are trying to beat 1192
15.4 Shortest Paths and Matrix Multiplication 1192
15.4.1 The dynamic-programming recipe, recalled 1192
15.4.2 Step 1: the structure of a shortest path 1192
15.4.3 Step 2: the recursive definition of $lij(m)$ 1193
15.4.4 The recurrence, term by term 1194
15.4.5 Why the first argument can be thrown away 1195
15.4.6 When does the sequence stop changing? 1196
15.4.7 Step 3: computing bottom up — EXTEND-SHORTEST-PATHS 1197
15.4.8 The analogy with ordinary matrix multiplication 1198
15.4.9 The (min, +) semiring, and why the analogy is exact 1199
15.4.10 SLOW-ALL-PAIRS-SHORTEST-PATHS 1199
15.4.11 The running example, every intermediate matrix 1200
15.4.12 One matrix computed cell by cell 1204
15.4.13 Where the matrices came from, in one picture 1205
15.5 Improving the Running Time: Repeated Squaring 1205
15.5.1 We only want the last matrix 1205
15.5.2 Associativity is what we sell 1206
15.5.3 The sequence of squares 1206
15.5.4 Why $\lceil \lg(n-1) \rceil$ products are enough 1207
15.5.5 FASTER-ALL-PAIRS-SHORTEST-PATHS 1207
15.5.6 The example again, in two products 1208
15.5.7 What repeated squaring does *not* give us 1210
15.6 The Floyd–Warshall Algorithm 1210
15.6.1 A different way to cut a path 1210

15.6.2 The two cases 1211
15.6.3 Why both halves are restricted to $1, \ldots, k-1$ 1212
15.6.4 The recurrence 1212
15.6.5 The algorithm 1214
15.6.6 Why k must be the outermost loop 1214
15.6.7 The predecessor matrix 1217
15.6.8 The running example, every $D(k)$ and $\Pi(k)$ 1218
15.6.9 The round $k = 1$, cell by cell 1224
15.6.10 The round $k = 4$, where most of the work happens 1225
15.6.11 Reading routes out of Π 1225
15.6.12 The running time 1226
15.6.13 Doing it in place, and why that is safe 1226
15.6.14 Detecting negative-weight cycles for free 1227
15.6.15 All four methods agree 1229
15.7 Transitive Closure of a Directed Graph 1233
15.7.1 The question, and the lazy answer 1233
15.7.2 The better answer: swap the operators 1233
15.7.3 The algorithm and the example 1234
15.7.4 Why booleans are so much faster in practice 1237
15.8 Johnson's Algorithm for Sparse Graphs 1239
15.8.1 The wish 1239
15.8.2 What a reweighting must accomplish 1239
15.8.3 The reweighting that works 1240
15.8.4 Choosing h: the auxiliary vertex 1241
15.8.5 The algorithm 1241
15.8.6 A small reweighting worked in full 1242
15.8.7 The five-vertex running example, reweighted 1244
15.9 Which Algorithm, and When 1248
15.9.1 The four costs side by side 1248
15.9.2 The crossover, measured 1249
15.9.3 A decision procedure 1252
15.10 Profitina in Practice 1252
15.10.1 Where all-pairs distances appear in a GEO platform 1252
15.10.2 The worked case study: Profitina Laundry 1253
15.11 Common Mistakes, and How to Avoid Them 1258
15.12 Chapter Summary 1259
15.12.1 Looking ahead 1260
15.13 Exercises 1260
15.14 Solutions to the Exercises 1261
References 1268

CHAPTER 16 Evaluation and Discussion IV: Shortest Paths, Amortized Analysis, and the Whole Book 1269
16.1 Where We Stand 1269
16.1.1 What Chapters 13 to 15 gave us 1269
16.1.2 The whole book on one page 1270
16.1.3 How this chapter is organized 1271
16.2 How to Approach a Multi-Part Problem or Proof 1272
16.2.1 Write the answer in a fixed order 1272
16.2.2 When a proof will not come out 1272
16.2.3 Five habits worth keeping 1273
16.3 Problem 16.1 — Fifteen Statements, True or False 1273
16.3.1 The problem 1273
16.3.2 What it tests 1274
16.3.3 The answers 1274

16.3.4 Statement (1) — the one that needs a sentence of qualification 1275
16.3.5 Statements (3), (4) and (7) — sortedness and the algorithm that uses it 1276
16.3.6 Statement (5) — when not to sort 1276
16.3.7 Statement (8) — the trap with two halves 1276
16.3.8 Statements (9), (10) and (12) — guaranteed against typical 1277
16.3.9 Statement (6) — why distinct weights force a unique MST 1277
16.3.10 Statements (11), (14) and (15) — bounds, counts and definitions 1278
16.3.11 Statement (13) — the one you can refute in five seconds 1279
16.3.12 The mistake most often made on this problem 1280
16.4 Problem 16.2 — The Longest Monotonically Increasing Subsequence 1281
16.4.1 The problem 1281
16.4.2 What it tests 1281
16.4.3 Assumptions, stated properly 1281
16.4.4 Deriving the recurrence 1281
16.4.5 The pseudocode, read line by line 1282
16.4.6 A complete worked example, by hand 1283
16.4.7 The time complexity, counted honestly 1284
16.4.8 The program 1285
16.4.9 A second method: $O(n\lg n)$ 1286
16.4.10 The mistakes most often made on this problem 1287
16.5 Problem 16.3 — Extended Binary Trees, Proved by Induction 1287
16.5.1 The problem 1287
16.5.2 What it tests 1288
16.5.3 The proof 1288
16.5.4 A tempting but careless write-up, and why it loses 1289
16.5.5 Two alternative proofs 1289
16.5.6 The experiment 1290
16.5.7 Why the result matters 1292
16.5.8 The mistakes most often made on this problem 1292
16.6 Problem 16.4 — A 2-Approximation for the Traveling Salesman 1292
16.6.1 The problem 1292
16.6.2 What it tests, and the vocabulary you need from nothing 1293
16.6.3 Part 1 — the algorithm produces a tour 1294
16.6.4 Part 2 — the running time is polynomial 1294
16.6.5 Part 3 — the ratio bound is 2 1295
16.6.6 What happens without the triangle inequality 1296
16.6.7 A complete worked instance 1296
16.6.8 The program 1297
16.6.9 How tight is 2? 1300
16.6.10 The mistakes most often made on this problem 1301
16.7 Problem 16.5 — Bellman–Ford against Dijkstra 1301
16.7.1 The problem 1301
16.7.2 What it tests 1301
16.7.3 Part 1 — the differences, and the applications 1302
16.7.4 Part 2 — the two pieces of pseudocode 1303
16.7.5 Part 3 — the illustrations 1304
16.7.6 The program 1306
16.7.7 Showing *why* Dijkstra fails, not just *that* it does 1308
16.7.8 The mistakes most often made on this problem 1310
16.8 Worked Problems on Single-Source Shortest Paths 1311
16.8.1 Problem 16.6 — Dijkstra, with the whole table after every iteration 1311
16.8.2 Problem 16.7 — Bellman–Ford, and a negative cycle found and exhibited 1314
16.8.3 Problem 16.8 — shortest paths in a directed acyclic graph 1319
16.9 Worked Problems on Amortized Analysis 1322
16.9.1 Problem 16.9 — the binary counter, three ways and by induction 1323

16.9.2 Problem 16.10 — the dynamic table, three ways, and a tempting wrong potential 1327
16.10 Worked Problems on All-Pairs Shortest Paths 1331
16.10.1 Problem 16.11 — Floyd–Warshall, every matrix computed cell by cell 1332
16.10.2 Problem 16.12 — Johnson's reweighting, carried out in full 1335
16.11 Further Worked Problems 1340
16.11.1 Problem 16.13 — fifteen more statements 1340
16.11.2 Problem 16.14 — the longest path in a DAG 1341
16.11.3 Problem 16.15 — a tree with n vertices has $n - 1$ edges 1343
16.11.4 Problem 16.16 — Huffman coding, in three parts 1344
16.11.5 Problem 16.17 — Floyd–Warshall against Johnson 1345
16.11.6 Problem 16.18 — fifteen statements again 1348
16.11.7 Problem 16.19 — the 0/1 knapsack, and what "polynomial" means 1349
16.11.8 Problem 16.20 — a 2-approximation for vertex cover 1352
16.11.9 Problem 16.21 — Prim against Kruskal 1354
16.12 The Problems of This Chapter Compared 1355
16.13 Quick Self-Check Drills 1357
16.14 The Algorithm Selection Table 1359
16.15 Where to Go Next 1361
16.16 Profitina in Practice 1363
16.16.1 The worked case study: Profitina Laundry 1364
16.17 Common Mistakes, and How to Avoid Them 1368
16.18 Chapter Summary, and a Closing Word 1369
A closing word 1370
16.19 Exercises 1370
16.20 Solutions to the Exercises 1371
References 1376

List of Figures

Figure 1.1 The goals of data structure and algorithm design (correctness, efficiency) set beside the goals of implementation (robustness, adaptability, reusability). 4
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. 6
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. 17
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. 18
Figure 1.5 *Journey*, E-Olymp problem 10280. Cossack Vus must visit every city in Potokolandia and come home again. 21
Figure 1.6 The two sample tests of *Journey*. Two cities at 1 and 4 give 6; three cities at 1,3,1 give 4. 22
Figure 1.7 *Unique Numbers*, SPOJ problem DCEPC901, with its two operation types and the crucial note that operations compose. 24
Figure 1.8 The constraints and the sample test of *Unique Numbers*: n up to 108, at most 20 operations. 25
Figure 1.9 Multiplication modulo 15. Each pair of rows shows the inputs 0..14 above and the outputs $(i \times x)$ mod 15 below, for $x = 3,5,2,4$ in turn. 26
Figure 1.10 *Minimum and maximum numbers*, SPOJ problem NEWLANG, with its constraints and sample test. 29
Figure 1.11 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. 31
Figure 1.12 *Interesting sequence*, E-Olymp problem 4894. The sequence is built by repeatedly appending a shifted copy of itself. 33
Figure 1.13 The sample test of *Interesting sequence*: three queries 17, 239, 385, with answers 1, 0, 2. 34
Figure 1.14 Indices 0 to 11 in binary, with the number of 1-bits and that count modulo 3. The last column is the sequence itself. 35
Figure 1.15 *Recursive function 1*, E-Olymp problem 10296. The definition, the single-integer input, and the sample $f(5) = 5$. 38
Figure 1.16 SPOJ problem PERIOD5, *Periodic function, trip 5*: given samples of a clean periodic function, extrapolate it. The small plot shows such a function, built from components of periods 3 and 5. 45
Figure 1.17 Paper [9] at a glance: title, byline, venue (*Journal of Theoretical and Applied Information Technology*, vol. 97, no. 8, 2019) and what it contributes. 45
Figure 1.18 Mean running time of three approaches as N grows, redrawn from the measurements in Figure 13 of [9]. The optimized $N \times N$ method stays flat; the others climb. 46
Figure 1.19 The SPOJ ranking for PERIOD5 as listed in early 2026. Of four hundred and twenty-four submissions only two hundred and thirty-five were accepted, and the ITS entry sits at rank 2 with a time of 0.59 seconds in 4.9 MB. 46
Figure 1.20 SPOJ problem HIR, *Hard Image Recognition*: recover a six-digit number from a black-and-white raster image, given a 0.1-second limit and a 150 kB source-code limit. 47
Figure 1.21 Ten six-digit numbers drawn in different fonts, some tilted, blurred, struck through, speckled or inverted — the kind of variety the HIR sample images show. 47
Figure 1.22 The sample test of HIR. The two pictures encode 000482 and 567833; the submission shown scores 1 because its second answer has one incorrect digit. 48
Figure 1.23 Paper [10] at a glance: title, byline, venue (*Journal of Theoretical and Applied Information Technology*, vol. 98, no. 17, 2020) and what it contributes. 49
Figure 1.24 The SPOJ ranking for HIR as listed in early 2026. Twenty-two users accepted out of six hundred and sixty submissions; the two top entries, scoring 108, are from the ITS group. 49
Figure 2.1 Muhammad ibn Musa al-Khwarizmi (c. 780–850 CE), whose Latinized name *Algorismus* became the word *algorithm*. 64

Figure 2.2 Charles Babbage (1791–1871), designer of the Analytical Engine, the first general-purpose computing machine ever conceived. 65
Figure 2.3 Ada Lovelace (1815–1852), who wrote the first algorithm intended to be executed by a machine. 66
Figure 2.4 Alan Turing (1912–1954), who formalized computation with the Turing machine in 1936. 66
Figure 2.5 Alonzo Church (1903–1995), inventor of the lambda calculus, who shares his name with Turing in the Church–Turing thesis. 66
Figure 2.6 John von Neumann (1903–1957), whose stored-program architecture underlies essentially every computer in use today. 67
Figure 2.7 The algorithm is an idea; the program is what that idea becomes on a particular machine, in a particular language. 69
Figure 2.8 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. 70
Figure 2.9 The sorting problem. The output $b1, \ldots, bn$ must be both *ordered* ($b1 \leq b2 \leq \cdots \leq bn$) 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. 75
Figure 2.10 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. 75
Figure 2.11 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. 84
Figure 2.12 Growth of $\lg n$, n, n, $n\lg n$, $100n$, $n2$ and $n3$ with both axes logarithmic. On such a plot a function nk becomes a straight line whose slope is k, so steeper means worse. 94
Figure 2.13 The same growth rates with $2n$ added. The vertical axis now runs to about 10308; every polynomial curve is crushed into the bottom of the chart, indistinguishable from the axis. 94
Figure 2.14 The searching problem. On the instance $A = 2,5,4,10,7$ with $q = 5$ the answer is the index 2; with $q = 9$ the answer is NIL, because 9 does not occur. 96
Figure 2.15 The sorted searching problem. The one added word — *sorted* — in the input specification is what makes an exponential speed-up possible. 99
Figure 2.16 The five goals once more. Everything in this chapter served the two on the left — correctness, proved with loop invariants, and efficiency, derived by counting. 114
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. 123
Figure 3.2 Big-O as an upper bound. Beyond the threshold $n0$, the running time $f(n)$ never rises above $c \cdot g(n)$. What happens to the left of $n0$ is irrelevant to the claim. 127
Figure 3.3 Big-Ω as a lower bound. Beyond $n0$, the running time $f(n)$ never falls below $c \cdot g(n)$. 130
Figure 3.4 Big-Θ as a tight bound. Past $n0$ the function $f(n)$ is trapped in the corridor between $c1 \cdot g(n)$ and $c2 \cdot g(n)$, so f and g grow at the same rate. 131
Figure 3.5 The arithmetic series and its closed form, $i = 0ni = 1 + 2 + 3 + \cdots + n = n2 + n2$. 142
Figure 3.6 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 $n(n + 1)/2$. 142
Figure 3.7 The arithmetic series as area. The bars of heights $1,2, \ldots, n$ sit astride the line of slope 1; their total area $n(n + 1)/2$ is the triangle $n2/2$ plus the n half-squares that overhang it. 143
Figure 3.8 The inductive step written out: assume $S(k) = k(k + 1)/2$ for $1 \leq k \leq n - 1$, then unfold $S(n) = S(n - 1) + n$ and simplify to $n(n + 1)/2$. 144
Figure 3.9 The finite geometric series, $i = 0nai = 1 + a + a2 + \cdots + an = 1 - an + 11 - a$. 144
Figure 3.10 The Merge Sort recurrence as a two-case definition: $T(n) = \Theta(1)$ when $n = 1$, and $T(n) = 2T(n/2) + \Theta(n)$ when $n > 1$. 149
Figure 3.11 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). 150

Figure 3.12 The substitution method failing. Assuming $T(k) \leq ck2$ yields $T(n) \leq cn2 + n$, and $cn2 + n \leq cn2$ holds for **no** choice of $c > 0$. 152
Figure 3.13 The strengthened hypothesis $T(k) \leq c1k2 - c2k$ closes the induction: the extra $-c2n$ absorbs the leftover $+n$ whenever $c2 \geq 1$. 153
Figure 3.14 Iterating $T(n) = n + 3T(\lfloor n/4 \rfloor)$ three times, collecting the geometric series $n\sum(3/4)i$, and concluding $T(n) = O(n)$....156
Figure 3.15 Building the recursion tree for $T(n) = T(n/4) + T(n/2) + n2$. Each expansion replaces a $T(\cdot)$ leaf by a node carrying its own non-recursive cost and two new children. 157
Figure 3.16 The completed recursion tree for $T(n) = T(n/4) + T(n/2) + n2$. The level totals $n2$,$516n2$,$25256n2$, … form a geometric series of ratio $5/16 < 1$, summing to $\Theta(n2)$. 158
Figure 3.17 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 $\log 3/2n$ levels, so the total is $O(n\lg n)$. 159
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. 194
Figure 4.2 The first ten terms of the LDCSUMAB sequence (left), collapsed to the map from k to n (middle), and the inversion $k = n(n+1)2$, $n2 + n - 2k = 0$, $n = -1 + 1 + 8k2$ (right). 196
Figure 4.3 A compact LDCSUMAB solution beside the table it implements: `n = ceil((-1+sqrt(1+8k))/2)`, `r = n(n+1)/2`, `m = n + k - (r+1)`. 200
Figure 4.4 An alternative TOHU program with a separate branch `if (n==1) printf("%.11lf\n", 2.0/3.0);` and a general branch that evaluates $112 - 12(n+1)(n+2) + 23$. 211
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. 262
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. 264
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. 270
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. 271
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 $2 \rightarrow 4 \rightarrow 9$, which is one root-to-leaf path and no more. 272
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. 277
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. 278
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. 278
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. 279
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. 279
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. 280
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. 281
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. 287

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. 287
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. 288
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. 288
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 $\leq x$, amber those known to be $\geq x$. The pointers i and j begin just outside the array and close in; when they cross, the partition is finished and j is returned. 303
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 $2d$ subarrays of size $n/2d$ costs $2d \times n/2d = n$. There are $\lg n + 1$ levels, so the total is $\Theta(n\lg n)$. 311
Figure 5.19 The worst-case recurrence unrolled: $T(n) = T(1) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n) = k = 1n\Theta(k) = \Theta k = 1nk = \Theta(n2)$ 311
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 $\Theta(n2)$. 312
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 $\log 10n$ levels. The right spine, whose sizes shrink only by a factor of $10/9$, survives to depth $\log 10/9n$. 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 $n\log 10n$ and $n\log 10/9n$, and both are $\Theta(n\lg n)$. 313
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 $L(n) = 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) = 2L(n/2-1) + \Theta(n) = \Theta(n\lg n)$. 315
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, $\Theta(n) + \Theta(n-1) = \Theta(n)$. 316
Figure 5.24 A single lucky split, for comparison: one partition of n elements, costing $\Theta(n)$, yielding two pieces of size about $(n-1)/2$. 316
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. 353
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. 360
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. 366
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. 367
Figure 6.5 Case 2 of deletion: node 30 has only a right child, 40. Deleting 30 promotes 40 into its place. 367
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. 368
Figure 6.7 An AVL tree. Every node's two subtrees differ in height by at most one. 376
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. 377
Figure 6.9 The two rotations. The same five items — the nodes x and y and the subtrees $P1, P2, P3$ — 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. 382
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. 387
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. 388

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. 389
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. 390
Figure 6.14 Inserting 14 into an AVL tree. Every balance factor stays inside $-1,0,+1$, so no rotation is performed. 391
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. 392
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. 393
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. 394
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. 396
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())`. 402
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`. 403
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 $T0$ to $T3$ name the four subtrees the rotation will move. 408
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 $T1$ (rooted at 50) as its right child; 78 is unchanged as the right child of 62. 409
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. 435
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.** 436
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. 437
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. 438
Figure 7.5 The argument in outline. Left: consecutive Fibonacci numbers approach the golden ratio $\phi \approx 1.61803$, so $Fn \approx 1.6n$. Right: the call count $2Fn + 1 - 1$ of the naive procedure, on a logarithmic scale, runs parallel to $1.6n$ — the tree has about $1.6n$ leaves, and the running time is exponential. 440
Figure 7.6 The repair, in brief: compute Fn 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. 443
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. 450
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 $c22$ of the result, by the formula $ci, j = l = 1mai, l \cdot bl, j$. 452
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 $A1 \times A2 \times \cdots \times An$, where Ai is a $di - 1 \times di$ matrix. 453

Figure 7.10 The two key observations. First, the outermost bracket must split the chain at *some* position k with $i \leq k < j$. Second, whatever k is, the parenthesizations on either side of it must themselves be optimal — optimal substructure, and the justification for the recurrence. 457
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 $di - 1dkdj$ for the last multiplication. 458
Figure 7.12 The count of the subproblems: there are only $n2 + n = \Theta(n2)$ different subproblems (i, j) with $1 \leq i \leq j \leq n$. The triangle draws them for $n = 6$: the 21 cells on and above the diagonal. That gap between $\Theta(n2)$ questions and exponentially many recursive visits is the opening dynamic programming exploits. 459
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. 462
Figure 7.14 The running time of **Matrix-Chain-Order**: easily seen to be $O(n3)$, and in fact also $\Omega(n3)$, hence $\Theta(n3)$. The plot, on a logarithmic scale, sets its candidate-split count against the $3n - 1$ 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. 463
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. 472
Figure 7.16 Step 2: write the recurrence. Its general form is $Mopt =$ minchoices k(sum of $Mopt$ of the subproblems produced by k) + (cost of making choice k) — and then show the number of distinct subproblem instances is bounded by a polynomial. 473
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. 474
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*. 475
Figure 7.19 The optimal substructure of LCS. Working from the *ends* of Xm and Yn: 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. 476
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. 477
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? 479
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. 530
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. 537
Figure 8.3 A tree that **is** an AVL tree. Every node's two subtrees differ in height by at most one. 547
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. 547
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. 556
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. 557
Figure 8.7 The two rotations. In both pictures the inorder reading is $P1, x, P2, y, P3$, 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. 558

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. $P1$ to $P4$ are the four subtrees. 559
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. 559
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**. 562
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**. 562
Figure 8.12 The first deletion case: the node to be deleted is a leaf. Simply remove it. 563
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. 563
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. 564
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. 566
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. 566
Figure 9.1 The eleven activities of the running example, with start times si and finish times fi, already sorted so that $f1 \leq f2 \leq \cdots \leq f11$. This one table is the running example for the whole of §9.4. 636
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 Aij would give more than Aij compatible activities — contradicting the optimality of Aij. 639
Figure 9.3 The reduction to a single subproblem: once the greedy choice $a1$ has been made, the single subproblem $S1$ remains, and an optimal solution to the original problem is $a1$ together with an optimal solution to $S1$. 641
Figure 9.4 Recursive-Activity-Selector in pseudocode. The **while** loop of line 2 finds the first activity in Sk to finish; the call is made with $k = 0$, using the fictitious activity $a0$ with $f0 = 0$. 644
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 ak (amber when it is taken), the teal bars are activities already selected, and the arrow marks the comparison sk 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 $a1, a4, a8, a11$ occupying the resource from time 1 to time 16. 645
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. 647
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. 665
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 $a = 0, b = 101, \ldots, f = 1100$. 667
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. 669
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 final tree. 670
Figure 9.11 The definition of a matroid: a finite ground set S, a hereditary family $\mathcal{I}$ of *independent* subsets, and the exchange property. 682

Figure 9.12 The graphic matroid $MG = (SG, \mathcal{I}G)$ 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. MG is closely related to the minimum-spanning-tree problem, which Chapter 11 covers in detail. 683

Figure 9.13 An instance of the problem of scheduling unit-time tasks with deadlines and penalties on a single processor: seven tasks $a1, \ldots, a7$ with deadlines di and penalties wi....688

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....721

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. 722

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. 723

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....725

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....728

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....728

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....729

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....730

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....730

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. 732

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. 737

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. 738

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 $2E$ — the handshake lemma, made visible. 739

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. 740

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. 749

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 $[w, r]$. 752

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. 752

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. 752

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. 753

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. 753

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. 765

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. 767

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. 768

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 $12 = 2V$. 768

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. 776

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: $(s\ (z\ (y\ (x\ x)\ y)\ (w\ w)\ z)\ s)\ (t\ (v\ v)\ (u\ u)\ t)$. Every pair of bars is either nested or disjoint; none overlap partially. 776

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**). 781

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. 791

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 $2V$ requires. The watch has no arrows at all: it may be put on at any time. 793

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. 793

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. 822
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. 826
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 11.3**. 828
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. 838
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. 841
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. 842
Figure 11.7 Figure 11.6 annotated: the two pieces left after removing the dotted edge (u, v) are named $T1$ (left) and $T2$ (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. 842
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. 844
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. 850
Figure 11.10 Step 2: the light edge for the cut $S = a$ is (a, b) of weight 4, so b joins the tree. 850
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). 851
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. 851
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. 851
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. 852
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. 852
Figure 11.16 Step 8: (c, d) of weight 7 joins d, and e's key improves from 10 to 9. 853
Figure 11.17 Step 9 — the final 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. 853
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. 860
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. 867
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. 867
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. 868

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. 869

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). 870

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. 870

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. 871

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. 899

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. 902

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. 903

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. 912

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 Akj — 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. 926

Figure 12.6 The eleven activities of §9.4.2 with their start and finish times, already sorted so that $f1 \leq f2 \leq \cdots \leq f11$. 927

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. 931

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. 932

Figure 12.9 The instance of §9.8.7 for scheduling unit-time tasks with deadlines and penalties: seven tasks with deadlines di and penalties wi. The greedy matroid algorithm schedules early a set of weight 230 and pays a penalty of 50. 939

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. 948

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. 950

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 $f[u]$. 951

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. 953

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. 953

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. 954

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. 971

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*....971

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. 983

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. 986

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. 990

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 12.35**, where a union changes exactly one pointer. 992

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. 1003

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. 1004

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. 1029

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 — here a, b, c and d — has $\delta(s, \cdot) = -\infty$. The problem is not hard on this graph; it is *ill-posed*. 1031

Figure 13.3 The two outcomes of `RELAX(u,v,w)` with $w(u, v) = 2$. **Top:** $d[u] + w = 5 + 2 = 7$, which is less than the current $d[v] = 9$, so $d[v]$ drops to 7 and $\pi[v]$ becomes u. **Bottom:** $d[u] + w = 7$, which is *not* less than the current $d[v] = 6$, so nothing at all happens. A relaxation that changes nothing is not a wasted step; it is a confirmation that the current estimate already beats this route. 1034

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. 1038

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. 1039

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 ∞. $Q = s = 0$, $u = \infty$, $v = \infty$, $x = \infty$, $y = \infty$. 1042

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. 1042

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) — the bold arrow now comes up from x. y is amber: at 7 it is the smallest key left. 1043

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. 1043

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. 1044

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 $G\pi$ — the shortest-path tree rooted at s. Reading a route off it is a matter of walking the bold arrows backward from any vertex to s. 1044

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 $s \rightsquigarrow p1x \to y \rightsquigarrow p2u$ 1049

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. 1056

Figure 13.14 The Bellman–Ford example, panel (a): initialization. Three of the ten edges carry negative weights — $w(x, t) = -2$, $w(y, x) = -3$, $w(t, z) = -4$ — which is exactly why Dijkstra may not be used here. There is no negative *cycle*, however: check $t \to z \to s \to t$, which costs $-4 + 2 + 6 = 4 > 0$, and $t \to x \to t$, which costs $5 + (-2) = 3 > 0$. 1058

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. The bold arrows are the current predecessor edges. 1059

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. 1059

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. This improvement was invisible until pass 3 because it needs three edges. 1060

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. 1060

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. 1067

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). Every arrow now points left-to-right or downward, so the graph is a DAG — and it still contains two negative edges, which the DAG algorithm handles without complaint. 1068

Figure 14.1 Operations $op1, \ldots, op7$ with their **actual costs** ti and the **amortized costs** ai we assign to them. The requirement is $\sum ti \leq \sum ai$ over the whole sequence, and emphatically **not** $ti \leq ai$ for each i 1091

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. 1093

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. 1095

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. 1099

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 $2 \times 16 = 32$. 1100
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. 1101
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. 1105
Figure 14.8 The complete solution to Exercise 17.1-3, from the split sum down to Total $\leq 3n - \lfloor \lg n \rfloor - 2$, with the geometric-series formula and the logarithm identity it needs. 1106
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 $O(n)$. 1113
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. 1118
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. 1122
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 $k' = \min(k, s)$. 1124
Figure 14.13 The counter by the potential method: with bi = the number of 1 bits after the i-th `Increment` and ti = the number of bits that increment reset, the potential change is at most $1 - ti$ and the amortized cost is at most $(ti + 1) + (1 - ti) = 2$. 1125
Figure 14.14 `Enqueue` and `Dequeue` for the two-stack queue: `Enqueue` is one push, and `Dequeue` pops from $SOUT$ after first draining all of SIN into it, **only if** $SOUT$ **is empty**. The underflow test comes after the transfer. 1129
Figure 14.15 The potential analysis of the two-stack queue. With $\Phi(Di) = 2 \times \text{elements}i$, where $\text{elements}i$ counts the items in the **input** stack only, `Enqueue` costs 3 amortized and both kinds of `Dequeue` cost 1. 1130
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. 1135
Figure 14.17 The aggregate analysis of `TABLE-INSERT`. One unit for every insertion, plus the expansion costs $20 + 21 + \cdots + 2\lfloor \lg n \rfloor$, which is less than $2n$; total under $3n$. 1136
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 $numi$ and the potential collapses by $numi - 3$. 1138
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. 1140
Figure 15.1 The input to every algorithm in this chapter: the $n \times n$ adjacency matrix $W = (wij)$ of Equation (15.1). Negative-weight edges are allowed; negative-weight cycles are not. 1185
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. 1186
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. 1187
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. 1193
Figure 15.5 The definition of $lij(m)$ and its base case $lij(0)$. Notice that the superscript is in parentheses — it counts edges, it is not an exponent. 1194
Figure 15.6 The recurrence of Equation (15.4), its one-line simplification, and the fact, Equation (15.6), that the sequence $lij(m)$ stops changing at $m = n - 1$. 1195
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 $\Theta(n3)$ time. 1197

Figure 15.8 SQUARE-MATRIX-MULTIPLY(A, B), the $\Theta(n3)$ schoolbook algorithm for $C = A \cdot B$. Set it beside **Figure 15.7** and the two differ in exactly two places. 1198
Figure 15.9 SLOW-ALL-PAIRS-SHORTEST-PATHS(W). It performs $n - 2$ calls to EXTEND-SHORTEST-PATHS, each costing $\Theta(n3)$, for a total of $\Theta(n4)$. 1200
Figure 15.10 The five-vertex graph (Figure 25.1 in the third edition of Cormen et al.) 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 $L(4)$. 1201
Figure 15.11 Only $\lceil \lg(n - 1) \rceil$ products are needed, because each one doubles the edge budget. The final product $L(2\lceil \lg(n - 1) \rceil)$ equals $L(n - 1)$ by Equation (15.6). 1206
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**. 1207
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. 1211
Figure 15.14 Path p (Figure 25.3 in the third edition of Cormen et al.) is a shortest path from i to j and k is its highest-numbered intermediate vertex. Both halves $p1$ and $p2$ then have all their intermediate vertices in $1,2, \ldots, k - 1$. 1212
Figure 15.15 The definition of $dij(k)$, the base case $dij(0) = wij$, and recurrence of Equation (15.7). The last matrix $D(n) = (dij(n))$ is the answer. 1213
Figure 15.16 FLOYD-WARSHALL(W): five words of state and three nested loops. Line 7 is recurrence of Equation (15.7) written out. 1214
Figure 15.17 The complete sequence of matrices $D(k)$ and $\Pi(k)$ computed by Floyd–Warshall for the graph of **Figure 15.2** (Figure 25.4 in the third edition of Cormen et al.). Row $k = 0$ is the input; row $k = 5$ is the answer. 1219
Figure 15.18 The base case $tij(0)$ and the recurrence for $tij(k)$, which is the Floyd–Warshall recurrence with $\vee$ for min and $\wedge$ for $+$. 1234
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 $\Theta(n3)$ time. 1235
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 (Figure 25.5 in the third edition of Cormen et al.). ... 1235
Figure 15.21 If a new weight function w preserves shortest paths and is non-negative everywhere, then $V \times$ Dijkstra on w answers the original question. 1239
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. 1242
Figure 15.23 A four-vertex graph with one negative edge, $0 \to 1$ of weight -5. Dijkstra is not allowed here. 1243
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. 1243
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. 1244
Figure 15.26 Johnson's algorithm on the graph of **Figure 15.2** (Figure 25.6 in the third edition of Cormen et al.). **(a)** G' with the original weights; the new vertex s is navy and each vertex v shows $h(v) = \delta(s, v)$. **(b)** After reweighting with $w(u, v) = w(u, v) + h(u) - h(v)$; every weight is now non-negative. **(c)–(g)** Dijkstra run from each vertex in turn, the source navy and the shortest-paths tree drawn in bold teal; each vertex shows $\delta(u, v)$ and $\delta(u, v)$ separated by a slash, related by $duv = \delta(u, v) = \delta(u, v) + h(v) - h(u)$. 1248
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. 1288
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. 1293
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. 1303

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. 1303
Figure 16.5 Dijkstra's example at initialization: $d[s] = 0$, and $d[u] = d[v] = d[x] = d[y] = \infty$. Edge weights $s \to u = 10$, $s \to x = 5$, $u \to v = 1$, $u \to x = 2$, $x \to u = 3$, $x \to v = 9$, $x \to y = 2$, $y \to v = 6$, $y \to s = 7$, $v \to y = 4$. 1304
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. 1305
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. 1305
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. 1306
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 $2 \times 16 = 32$. 1324

List of Tables

Table 1.1 The data structures used in this book, and the single thing each one is best at. 4
Table 1.2 The common growth rates, and what each one costs at $n = 1000$. 9
Table 1.3 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 108 budget to them. 10
Table 1.4 The five design paradigms, the single move that characterizes each, and where each appears in this chapter. 15
Table 1.5 The five phases. Nothing here is optional, and the order is not negotiable. 15
Table 1.6 The memoization table built while computing $f(12)$, in the order the entries were filled. 41
Table 1.7 The six problems of this chapter, their paradigms, and the single sentence that solves each. 42
Table 1.8 Symptoms and diagnoses. When a problem statement contains a phrase on the left, the technique on the right is worth trying first. 43
Table 1.9 The four online judges met in this book. Their sizes change every day; the figures are those reported on the dates shown. 44
Table 1.10 The two families of books behind this one, with the editions current in September 2026. 50
Table 1.11 The sixteen chapters of this book, with the corresponding chapters of [11]. 50
Table 1.12 The four evaluation chapters and what each one covers. 51
Table 1.13 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. 52
Table 2.1 Insertion Sort on $A = 5,2,4,6,1,3$, one outer iteration per row. The bar between the sorted prefix and the untouched suffix moves one place right on every row. 77
Table 2.2 The cost of each line of Insertion Sort and the number of times it runs. The symbol tj is the number of times the line-4 test is evaluated for that value of j 83
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. 92
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. 95
Table 2.5 Binary search for $q = 7$ in $A = 2,4,5,7,10$. Two comparisons suffice where linear search would need four. 100

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. 107
Table 2.7 The topics of this book mapped onto *Introduction to Algorithms*, 4th edition (computational geometry appears only in the 3rd edition). This chapter is rows one and two. 108
Table 2.8 Worst-case comparison counts for binary search on n elements. 118
Table 3.1 The growth classes, what happens to each when the input size doubles, and where each one shows up in this book. 126
Table 3.2 The five asymptotic notations and the numerical comparison each one imitates. 133
Table 3.3 The logarithm identities used in this chapter. The last two are the ones that make the Master Theorem work. 137
Table 3.4 The common functions in increasing order of growth. Each row is $o(\cdot)$ of every row below it. 141
Table 3.5 The summations used throughout this book, with closed forms and asymptotic classes. 147
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. 160
Table 3.7 The three cases of the Master Theorem for $T(n) = a\,T(n/b) + f(n)$, with the intuition each one expresses. 164
Table 3.8 All twenty worked problems of §3.14 at a glance, with the case each falls into. 175
Table 3.9 Everything in Chapter 3, on one page. 186
Table 3.10 The algorithms met so far, their recurrences, and how each is solved. 186
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. 192
Table 4.2 The first ten numbers of the form $2m + 2n$, computed by hand. The column n reads one 1, two 2s, three 3s, four 4s; inside each block m counts 0,1,2, … 196
Table 4.3 The first five terms of the TOHU sequence, recovered by hand from the weighted sums. From $k = 2$ on, ak is the reciprocal of three consecutive integers; $a1$ breaks the pattern. 206
Table 4.4 Horner's rule on $2x3 - 6x2 + 2x - 1$ at $x = 3$. After each iteration, p equals the partial polynomial predicted by the invariant; after the last, it equals $P(3) = 5$. 215
Table 4.5 Insertion Sort on 8,3,6,3,9,1. The total number of shifts, 9, equals the number of inversions; tj counts executions of the `while` test on line 4 and is always one more than the shifts. 218
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. 220
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 $4 \cdot 5 \cdot 66 = 20$. 223
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. 228
Table 4.9 The recursion tree of $T(n) = 3T(n/4) + n2$, level by level. Each level costs 316 of the level above, so the root dominates. 233
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. 236
Table 4.11 Fast doubling computing $(F6, F7) = (8,13)$ in four calls. Each row checks one instance of the doubling identities. 241
Table 4.12 Measured times from **Listing 4.17**. Doubling n multiplies the brute-force time by about 4, as $\Theta(n2)$ predicts, and the sort-and-search time by about 2, as $\Theta(n\lg n)$ predicts; the advantage itself doubles with every row. 245
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. 247
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. 258
Table 5.2 Moving around the hidden tree with arithmetic alone. Every one of these is a single machine instruction. 263
Table 5.3 The cost of Heapsort, line by line. The dominant term is the $n - 1$ calls to `MAX-HEAPIFY`. 289
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. 295

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. 301
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. 302
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. 327
Table 5.8 Choosing a sorting algorithm. The last row is the one you will use most often in practice. 334
Table 5.9 Heights for Exercise 5.2. The height increases by one exactly at each power of two, which is what the floor of a base-two logarithm does. 344
Table 5.10 The Lomuto partition of Exercise 5.11, iteration by iteration. 346
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. 354
Table 6.2 The three depth-first traversals and the jobs each one does. 359
Table 6.3 The three cases of deletion in a binary search tree, and the work each one costs beyond the initial search. 369
Table 6.4 The fewest nodes an AVL tree of height h can contain, and the Fibonacci number it equals once you add one. 379
Table 6.5 The four rebalancing cases, identified by two balance factors, with the rotations each one needs. 391
Table 6.6 Choosing an ordered dictionary. k is the number of keys the range query actually returns. 414
Table 6.7 Worst-case costs for a plain binary search tree and for an AVL tree with n nodes. 423
Table 7.1 The sliding window of `fibIter` computing $F6$. After the last iteration `cur` holds $8 = F6$, and no array was ever allocated. 445
Table 7.2 The five ways to compute Fn 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. 449
Table 7.3 The finished M table for $d = 10,20,3,5,30$. 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, $M[1,4] = 1950$. 461
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. 461
Table 7.5 Memoization against bottom-up. Both give the same asymptotic running time; everything that differs is in this table. 472
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. 481
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. 481
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. 490
Table 7.9 The c table for Exercise 7.10. Reconstruct the bracketing it encodes. 497
Table 7.10 The sliding window computing $F8 = 21$. Two variables, seven iterations, no array. 499
Table 7.11 The M table for Exercise 7.7. The c table is $c[1,2] = 1$, $c[1,3] = 1$, $c[1,4] = 3$, $c[1,5] = 3$, $c[1,6] = 3$; $c[2,3] = 2$, $c[2,4] = 3$, $c[2,5] = 3$, $c[2,6] = 3$; $c[3,4] = 3$, $c[3,5] = 3$, $c[3,6] = 3$; $c[4,5] = 4$, $c[4,6] = 5$; $c[5,6] = 5$. 500
Table 7.12 The c table for Exercise 7.11. The LCS length is $c[7,6] = 4$. 501
Table 7.13 The c table for Exercise 7.12. The LCS length is 2, and three distinct LCSs achieve it. 502
Table 7.14 The edit-distance table for "kitten" and "sitting". The answer, $e[6,7] = 3$, is the bold cell. 503
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. 508
Table 8.2 The revision map. Every chapter before this one contributes something the problems of this chapter actually need. 509

Table 8.3 `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. 513
Table 8.4 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 $q = 5$. 520
Table 8.5 The correct recursion traced on the trap tree 20105,25,30. The bound $u = 20$ reaches node 25 through *two* levels of recursion; that is the information the local check throws away. 535
Table 8.6 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. 554
Table 8.7 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. 562
Table 8.8 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. 573
Table 8.9 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. 573
Table 8.10 The cost table M for $d = 4,10,3,12,20,7$. Each diagonal was filled before the next one; the answer is the top-right corner, $M[1,5] = 1\,344$. 584
Table 8.11 The split table c for the same chain. The two bold entries are the ones the reconstruction of §8.15.8 reads. 584
Table 8.12 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. 593
Table 8.13 Problems 8.18 to 8.25 against the problems they extend. The shape is the same; only the invariant changed. 599
Table 8.14 Exercise 1: the recursion stops at node 45, whose upper bound 40 came from the root two levels above. Nodes 60, 50 and 70 are never visited. 617
Table 8.15 Exercise 6: inserting 1 to 7 in ascending order. Every rotation is the Right-Right case, because ascending input always leans right, and four single rotations are enough to keep the height at 2 instead of 6. 618
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. 634
Table 9.2 The instance of **Figure 9.1** with the duration of each activity worked out. Activity $a10$ is the monster: twelve time units, nearly the whole day. 636
Table 9.3 Sorting plus one greedy sweep, measured by **Listing 9.2** 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. 648
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. 654
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. 656
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. 658
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. 659
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. 665
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. 671
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. 689
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". 694

Table 9.12 Conflict counts for the nine activities of Exercise 9.4, before any selection is made. Five activities tie at the minimum of two. 708
Table 10.1 In- and out-degrees of the directed graph used in §10.8. Both columns sum to $8 = E$, as the lemma requires. The self-loop $z \to z$ contributes one to the out-degree of z and one to its in-degree. 727
Table 10.2 How far apart sparse and dense really are. For a million vertices a matrix holds 1012 entries — a terabyte even at one byte each — while a sparse graph on the same vertices may have only a million edges. This single table is the reason §10.5 spends so long on representations. 735
Table 10.3 The complete vocabulary of §10.4, with an instance of each term in the graph of **Figure 10.1**. Every one of these words is used somewhere in the rest of the chapter. 736
Table 10.4 Space cost of the three representations. For the tiny graph of **Figure 10.1** the three are within a factor of two of each other, which is why small examples mislead. At a million vertices the matrix is off the table entirely. 741
Table 10.5 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. 742
Table 10.6 When to reach for which representation, and the reason in each case. If in doubt, choose the adjacency list; it is the right answer far more often than not. 742
Table 10.7 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. 753
Table 10.8 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. 754
Table 10.9 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. 768
Table 10.10 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. 769
Table 10.11 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. 782
Table 10.12 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. 797
Table 10.13 The measurements of **Listing 10.10**, with the growth ratios added. Absolute times depend on the machine and vary by a few tens of percent from run to run, so it is the *ratios* that carry the claim: doubling the graph roughly **doubles** the adjacency-list time and very nearly **quadruples** the matrix time — the signatures of $\Theta(V + E)$ and $\Theta(V2)$ respectively. The last two columns, the time divided by the predicted cost, stay within a small factor across an eight-fold range of sizes, which is what "linear" and "quadratic" mean in practice. 799
Table 10.14 All six simple paths from b to d in the graph of **Figure 10.1**, in order of increasing length. The shortest have length 2; there is none of length 1 because $(b, d) \notin E$. 812
Table 10.15 Breadth-first search from w on the graph of §10.7.4. Compare with **Table 10.7**: the same eight steps, but a different tree and different distances, because the source has moved. 813
Table 10.16 Why the breadth-first tree of §10.7.4 is independent of the adjacency ordering: every vertex has exactly one neighbor one level closer to s, so it has no choice of parent. 814
Table 10.17 Timestamps when the DFS outer loop scans z, y, x, w, v, u. Compare **Table 10.9**, which used the order u, v, w, x, y, z: the same graph, the same algorithm, an entirely different forest. 814
Table 10.18 The effect of one new road on Profitina Laundry's hop counts. One zone joins the two-hop promise and one leaves the four-hop tail; the zone with no road at all is unaffected, because adding an edge elsewhere cannot create a path to a different component. 818
Table 11.1 The four kinds of edge a depth-first search of a directed graph can traverse, classified by the color of the far endpoint at the moment the edge is examined. The whole of §11.2 turns on the second row. 821

Table 11.2 The nine precedence constraints of **Figure 11.2**, read off the arrows. Note what is *not* here: nothing relates `socks` to `shirt`, and nothing at all relates `watch` to anything. Those freedoms are why the answer is not unique. 827

Table 11.3 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. 828

Table 11.4 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. 833

Table 11.5 Choosing between the two topological-sort algorithms. Both are $\Theta(V + E)$ and both are correct; everything in this table is about engineering, not complexity. 837

Table 11.6 The two algorithms of this chapter are the same algorithm with two different answers to line 3.1 of `MoreSpecific-MST`. Everything else — the correctness proof, the $V - 1$ edge count, the loop invariant — is shared. 848

Table 11.7 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. 853

Table 11.8 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. 855

Table 11.9 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 final 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. 862

Table 11.10 How the inverse Ackermann function grows. The number of atoms in the observable universe is about 2266, so for every disjoint-set problem that will ever be run on physical hardware, $\alpha(n) \leq 4$. 863

Table 11.11 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. 871

Table 11.12 Choosing between the two algorithms. The first two rows cover most real cases; the rest are the cases where the answer is not obvious. 879

Table 11.13 The uniqueness test of §11.12.3 applied to the nine-vertex example graph, computed by calling Kruskal from **Listing 11.5** 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**. 881

Table 12.1 Every idea of Chapters 9 to 11, the section that teaches it, the kind of question it raises, and the place in this chapter where a problem of that kind is worked in full. 898

Table 12.2 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. 900

Table 12.3 Six candidate projects put through the load-bearing test. Four survive it. The two that do not are not bad programs; they are poor *design* problems, because they give the design and the evaluation nothing to argue about. 901

Table 12.4 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. 902

Table 12.5 The predicted cost of Rute Laundry, phase by phase, written before any code was compiled. The sorting step dominates, and §12.2.10 measures whether that is what actually happens. 904

Table 12.6 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. 905
Table 12.7 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. 906
Table 12.8 Kahn's algorithm on the ten pipeline stages. Ten stages emitted in ten rounds, so the specification is acyclic and the order is legal. In round 4 *invoice* is emitted before *pretreat*, not because it is urgent but because the queue is first-in-first-out and invoice was enqueued first. 907
Table 12.9 The twenty-three worked problems of Sections 12.3 to 12.6, the section of this chapter where each is worked, and the tool from Chapters 9 to 11 that solves it. 919
Table 12.10 The ten dryer jobs of Worked Problem B, listed in non-decreasing order of finish time. 924
Table 12.11 The ten dryer jobs with their durations and the number of other jobs each one clashes with. Jobs $a1$, $a2$, $a4$ and $a10$ tie for the fewest conflicts at two. 924
Table 12.12 The six jobs of Worked Problem C, with processing times and deadlines. 927
Table 12.13 Earliest deadline first on the six jobs. Only $J4$ is late, by one unit, so $L\max = 1$. 928
Table 12.14 Shortest processing time first on the same six jobs. $L\max = 6$, six times worse than earliest deadline first, even though the *total* completion time is smaller. 928
Table 12.15 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. 930
Table 12.16 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. 931
Table 12.17 The four items of Worked Problem E. 933
Table 12.18 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. 934
Table 12.19 The seven pressing orders of Worked Problem F: each takes one hour, and a late order costs its refund. 935
Table 12.20 The matroid greedy algorithm on the seven orders. Bold counts break the rule $Nt \leq t$. Two orders are rejected, and the refund paid is their sum, $30 + 20 = 50$. 936
Table 12.21 Breadth-first search of $G1$ 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. 949
Table 12.22 The output of breadth-first search on $G1$: distance and predecessor. Reading π backward from any vertex and reversing gives a shortest path, and its length always equals that vertex's d. 949
Table 12.23 Timestamps and predecessors after depth-first search of $G2$. The clock stopped at $14 = 2V$, and the fourteen stamps are the integers 1 to 14, each used once — the first check to run on any DFS answer. 952
Table 12.24 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. 952
Table 12.25 Depth-first search of the undirected graph $G1$ from A. The clock ends at $18 = 2V$. The intervals are perfectly nested, because the depth-first tree here is a single path through all nine vertices. 955
Table 12.26 The component label of every vertex of $G3$. Three components. Once this array exists, connectivity is a single integer comparison, which is why the array and not the search is the answer. 959
Table 12.27 The eight subjects of the prerequisite digraph used in Problems M to Q, with their durations and prerequisites. Nine arcs in all: $A \to D$, $B \to D$, $B \to E$, $C \to G$, $D \to E$, $D \to F$, $E \to H$, $F \to H$, $G \to H$. 969
Table 12.28 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. 970
Table 12.29 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. 972

Table 12.30 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. 981

Table 12.31 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. 982

Table 12.32 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. 984

Table 12.33 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. 985

Table 12.34 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. 991

Table 12.35 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. 992

Table 12.36 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**. 994

Table 12.37 Breadth-first search of $G1$ from C, beside the search from A. Every vertex changes its distance. The greatest distance to any vertex — the **eccentricity** of the source — is 4 from A and 3 from C, so C is the more central of the two and its tree is one level shallower. 1012

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 s, x, u, v. Enumerating paths is correct but hopeless in general: a graph on n vertices can have more than $(n-2)!$ simple paths between one pair. 1025

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. 1025

Table 13.3 The edge form of the triangle inequality checked on every edge of the Dijkstra example graph. The rows showing *equality* — (s, x), (x, u), (u, v), (x, y) — are precisely the four edges of the shortest-path tree, as §13.6.9 explains. 1030

Table 13.4 What each of the three algorithms does in each weight regime. The word to notice is *silently*: Dijkstra does not crash or complain on a negative edge, it returns a plausible-looking wrong number. §13.7.3 builds the smallest example. 1032

Table 13.5 The five properties of relaxation. They are proved once, here, and used by all three algorithms; no algorithm in this chapter needs a correctness argument that does not reduce to this table. 1037

Table 13.6 The complete run of Dijkstra on the example graph, one row per iteration of the `while` loop. The keys in Q only ever fall, while the extracted keys 0,5,7,8,9 never fall; §13.7.8 proves the second fact. 1044

Table 13.7 The output of Dijkstra on the example graph: distance, predecessor, and the route found by walking π backward. The d row agrees with **Table 13.1**, which found $\delta(s, v) = 9$ by listing six paths. 1045

Table 13.8 Dijkstra's running time for three priority-queue implementations. The binary-heap total simplifies to $O(E \lg V)$ when every vertex is reachable from s, so that $E \geq V - 1$; the Fibonacci-heap row is due to Fredman and Tarjan [5]. 1051

Table 13.9 Breadth-first search, Prim and Dijkstra compared. The single line 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 together, so it never needs the triangle inequality. 1055

Table 13.10 The complete run of Bellman–Ford on the example graph, one row per pass over all ten edges. In pass 2, $d[x]$ was set to 11 and then to 4 *within the same pass*; only the value at the end of each pass is shown. 1061

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), a structure a greedy method that finalizes vertices early can easily miss. 1061

Table 13.12 `DAG-Shortest-Paths` on **Figure 13.20**, one row per vertex in topological order. Compare with **Table 13.10**: Bellman–Ford needed four passes over ten edges — forty relaxations — to solve the full graph. Here seven relaxations suffice. Note also that the answers differ ($\delta(s,t)$ is 6 here and 2 there) because deleting (x,t) removed the cheap route; the two are different graphs, not two answers to the same question. 1069

Table 13.13 Which single-source algorithm to use. Read it top-down: first ask whether the graph is acyclic, then whether any weight is negative, then how dense it is. The first two questions are about *correctness* and admit no compromise; only the third is about speed. 1076

Table 14.1 The values $Ak(1)$ that define α. The jump from $k = 3$ to $k = 4$ is not a jump in size, it is a jump in *kind*: 2047 is a number you can write down, and $A4(1)$ is a number whose decimal expansion would not fit in the observable universe. 1151

Table 14.2 The inverse Ackermann function. For $\alpha(n)$ to reach 5, you would need more than $A4(1) > 22048$ elements. 1151

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. 1155

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. 1192

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. Everything else — the loop structure, the index pattern, the running time — is identical. 1198

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. 1210

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. 1249

Table 15.5 A decision procedure for the all-pairs shortest-paths problem. Work down the rows; the first "yes" is your answer. 1252

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. 1271

Table 16.2 The fifteen answers to Problem 16.1, with a one-line justification for each. Nine are true and six are false. 1275

Table 16.3 The dynamic-programming table for $S = 9,2,5,3,7,11,8,10,13,6$. $L[i]$ is the length of the longest increasing subsequence ending at $S[i]$; $P[i]$ is the index of the element before it. 1284

Table 16.4 The cost of each step of the approximation algorithm on a complete graph of n vertices. The minimum spanning tree dominates. 1294

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. 1296

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. 1302

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. 1306

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. 1312

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. 1315

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 $3 + (-2) = 1 < 2$. 1316

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. 1320

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). 1325

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 $i = 2,3,5,9$. 1329
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. 1332
Table 16.15 Every entry that changes during Floyd–Warshall on Problem 16.11, with the comparison that changed it. Note that $d32$ changes twice: a route found in round 1 is beaten by a route found in round 4. 1333
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. 1336
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. 1355
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. 1356
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. 1360
Table 16.20 The tables for Exercise 2. The maximum is $L[8] = 4$, so the backtrack starts at $k = 8$. 1372

List of Listings

Listing 1.1 Insertion sort in C++, complete and runnable, sorting the sequence from **Figure 1.2**. 7
Listing 1.2 A tracing version of insertion sort that prints the loop invariant after every pass. 8
Listing 1.3 The incremental paradigm at its simplest: one running variable, one pass. 10
Listing 1.4 Merge sort: split in half, sort each half, merge. The divide-and-conquer paradigm in twenty lines. 11
Listing 1.5 Naive recursion against memoized recursion for Fibonacci numbers. The call counter makes the difference impossible to argue with. 12
Listing 1.6 The greatest common divisor computed twice: by exhaustive search, and by Euclid's algorithm. 14
Listing 1.7 A brute-force verifier for the coloring constructed in the proof of *Columns*. 19
Listing 1.8 The complete solution to *Columns*, E-Olymp 10281. 21
Listing 1.9 The complete solution to *Journey*, E-Olymp 10280. 24
Listing 1.10 The complete solution to *Unique Numbers*, SPOJ DCEPC901. 27
Listing 1.11 The complete solution to *Minimum and maximum numbers*, SPOJ NEWLANG. 32
Listing 1.12 Building *Interesting sequence* by brute force and checking the popcount formula against every term. 36
Listing 1.13 The complete solution to *Interesting sequence*, E-Olymp 4894. 36
Listing 1.14 Counting the distinct sub-problems of f by exploring the recursion with a `set`. 39
Listing 1.15 The complete solution to *Recursive function 1*, E-Olymp 10296. 41
Listing 1.16 The first program: it proves that the compiler, the editor and the terminal all work together. 53
Listing 1.17 The Profitina Laundry planner: the van's shortest round trip and the number of rack labels required. 55
Listing 2.1 Insertion Sort in C++, with 1-based indexing so that the code matches the pseudocode exactly. 77
Listing 2.2 Insertion Sort instrumented to print its own trace, one line per outer iteration. 78
Listing 2.3 Counting $j = 2ntj$ for ascending, descending and random input, alongside the predicted worst-case value. 88

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. 90
Listing 2.5 Computing the growth-rate comparison, including 2106, without ever storing a number that large. 92
Listing 2.6 Linear search in C++, on the same data as **Figure 2.14**. 98
Listing 2.7 Binary search in C++, iterative form, instrumented to print the window at each step. 101
Listing 2.8 Why the midpoint must be computed as `left + (right - left) / 2`. 102
Listing 2.9 Binary search as a recursive divide-and-conquer procedure, printing the window at each level of recursion. 103
Listing 2.10 Profitina Laundry's counter book: incremental insertion to keep it ordered, binary search to find a ticket. 109
Listing 2.11 Insertion Sort with ordinary 0-based indexing and no unused element. 115
Listing 2.12 Verifying that Insertion Sort's shift count equals the input's inversion count. 117
Listing 2.13 Binary search for the first and last occurrence of a value, with duplicates present. 118
Listing 3.1 A C++ program that tabulates the standard growth functions. Save it as `growth.cpp`, compile with `g++ -std=c++17 -O2 -o growth growth.cpp`, and run `./growth`. 124
Listing 3.2 Hunting numerically for the constants c and $n0$ in the definition of Big-O. 128
Listing 3.3 The iterated logarithm, Stirling's approximation and the Fibonacci closed form, computed and compared. 139
Listing 3.4 Brute-force addition checked against every closed form derived in §3.9. 147
Listing 3.5 Evaluating two recurrences exactly, and comparing against the bounds proved by substitution. 154
Listing 3.6 Summing recursion trees level by level for the three shapes of **Table 3.6**, and checking the leaf-count identity. 160
Listing 3.7 A Master Theorem solver in C++, applied to eighteen recurrences. 171
Listing 3.8 Merge Sort in C++, with a printed trace and a measured comparison count. 178
Listing 3.9 Profitina Laundry: Merge Sort by promised pickup time, counting inversions in the same $\Theta(n\lg n)$ pass. 181
Listing 4.1 LDCSUMAB — unranking the sequence $2m + 2n$, in $O(1)$ per query after the square root. 200
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. 202
Listing 4.3 A brute-force cross-check for LDCSUMAB: the list built and sorted explicitly, compared with the formula at all 820 positions. 203
Listing 4.4 TOHU — Help Tohu, in $O(1)$ per test case, with no special case for $n = 1$. 211
Listing 4.5 TOHU from first principles: each ak recovered exactly from the weighted-sum condition, with no formula assumed, and compared with the closed form. 212
Listing 4.6 Horner's rule against naive evaluation, with a trace of the loop and a count of multiplications. 215
Listing 4.7 Insertion Sort with a per-iteration trace, and an independent inversion count for comparison. 218
Listing 4.8 Binary search with a line of output per probe, on the array of Worked Problem C. 221
Listing 4.9 Counting the executions of the innermost line of LOOPS and comparing with $n(n + 1)(n + 2)6$. 223
Listing 4.10 Numerical checks of the witnesses $c1$, $c2$, $n0$ for claims (a) and (d), and of the escape point in claim (c). 225
Listing 4.11 The binary logarithm of each of the thirteen functions at four sizes, showing where the asymptotic order does and does not yet hold. 228
Listing 4.12 Both recurrences of Worked Problem G evaluated exactly, with the proved bounds checked for every $n \leq 106$. 231
Listing 4.13 The recursion tree of $T(n) = 3T(n/4) + n2$ at $n = 256$, level by level, and the ratio $T(n)/n2$ as n grows. 234
Listing 4.14 The four recurrences of Worked Problem I evaluated exactly and divided by their predicted bounds. 236
Listing 4.15 LAUNDRT — Laundry Tokens, solved in $O(1)$ by telescoping. 239

Listing 4.16 STRIPTILE — tilings of a $2 \times N$ strip by fast doubling, $\Theta(\log N)$ per query. 242
Listing 4.17 Brute force against sort-and-search for the pair-sum problem, timed on inputs of doubling size. 244
Listing 4.18 Profitina Laundry — the most profitable contiguous stretch of days, by the prefix-minimum method, with the tie-breaking rule demonstrated on a second dataset. 249
Listing 5.1 Every structural fact about heaps from Section 5.3, computed and checked. 266
Listing 5.2 `MAX-HEAPIFY` in C++, narrating each of its own decisions. 274
Listing 5.3 `BUILD-MAX-HEAP`, traced on the array of Section 5.5.3 and then measured on random arrays up to ten million elements. 284
Listing 5.4 A complete, runnable Heapsort that narrates every extraction. 290
Listing 5.5 A complete max-priority queue, with all four operations exercised. 295
Listing 5.6 Heaps from the C++ standard library: `std::priority_queue`, custom comparators, and the raw `make_heap` / `sort_heap` algorithms. 298
Listing 5.7 Lomuto's and Hoare's partition schemes, each narrating its own steps. Indices here are 0-based, so subtract one from every index in the hand traces above. 304
Listing 5.8 Quicksort with Lomuto partitioning, narrating the recursion. Indentation shows the depth of the call. 308
Listing 5.9 Counting Quicksort's comparisons on sorted and shuffled input, with and without a random pivot, against the two theoretical formulas. 318
Listing 5.10 Measuring the recursion depth of naive and tail-recursive Quicksort on sorted and shuffled input. 322
Listing 5.11 The decision-tree bound computed for small n: the number of leaves required, and the resulting minimum height. 326
Listing 5.12 A stability test. Each record carries a tag showing where it started, so reordering among equal keys is visible. 328
Listing 5.13 A head-to-head benchmark of all four sorts on random data, with each sort given a fresh copy and its output checked. 331
Listing 5.14 Profitina Laundry: a heap decides the washing order, and Quicksort produces the takings report. 336
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`. 354
Listing 6.2 The three traversals in C++. Save as `traversal.cpp`, compile with `g++ -std=c++17 -O2 -o traversal traversal.cpp`, and run `./traversal`. 356
Listing 6.3 Search, minimum, maximum, successor and predecessor, each reporting the path it walked. Compile with `g++ -std=c++17 -O2 -o query query.cpp`. 362
Listing 6.4 A complete binary search tree. Compile with `g++ -std=c++17 -O2 -o bst bst.cpp` and run `./bst`. 369
Listing 6.5 Measuring what insertion order does to a binary search tree. Compile with `g++ -std=c++17 -O2 -o degenerate degenerate.cpp`. 373
Listing 6.6 The minimum-nodes recurrence, the Fibonacci identity and the height bound, all checked numerically. Compile with `g++ -std=c++17 -O2 -o minnodes minnodes.cpp`. 381
Listing 6.7 A right rotation with its exact pointer updates, applied to the left-leaning chain 30 → 20 → 10. Compile with `g++ -std=c++17 -O2 -o rotright rotright.cpp`. 383
Listing 6.8 A left rotation, applied to the degenerate chain 10 → 20 → 30. Compile with `g++ -std=c++17 -O2 -o rotleft rotleft.cpp`. 385
Listing 6.9 A complete AVL tree. Compile with `g++ -std=c++17 -O2 -o avl avl.cpp` and run `./avl`. 397
Listing 6.10 One insertion never costs more than one rebalancing; one deletion can cost a rebalancing at every level. Compile with `g++ -std=c++17 -O2 -o cascade cascade.cpp`. 404
Listing 6.11 Measuring an AVL tree against a plain BST, a red-black tree and a hash table. Compile with `g++ -std=c++17 -O2 -o compare compare.cpp`. 411
Listing 6.12 The Profitina Laundry pickup board: an AVL tree keyed by promised pickup time, supporting a successor query, a range query and deletion. Compile with `g++ -std=c++17 -O2 -o laundry laundry.cpp`. 416

Listing 6.13 The wrong and the right way to verify the binary-search-tree property. Compile with `g++ -std=c++17 -O2 -o isbst isbst.cpp`.....420
Listing 6.14 The k-th smallest key of a binary search tree in $O(h+k)$ time. Compile with `g++ -std=c++17 -O2 -o kth kth.cpp`.....429
Listing 6.15 An order-statistic tree: the AVL tree of **Listing 6.9** with a subtree size in every node, supporting `select` and `rank` in $O(\lg n)$. Compile with `g++ -std=c++17 -O2 -o ostree ostree.cpp`.....430
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`.....441
Listing 7.2 Memoized, bottom-up, constant-space and matrix-power Fibonacci, all four timed against each other. Compile with `g++ -std=c++17 -O2 -o fib_fast fib_fast.cpp`.....446
Listing 7.3 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`.....455
Listing 7.4 Matrix-Chain-Order and **Print-Optimal-Parens** in C++, with a full trace of every candidate split. Compile with `g++ -std=c++17 -O2 -o mcm mcm.cpp` and run `./mcm`.....464
Listing 7.5 The same matrix-chain recurrence implemented three ways — plain recursion, memoized recursion, bottom-up table — checked against each other and timed. Compile with `g++ -std=c++17 -O2 -o mcm_compare mcm_compare.cpp`.....469
Listing 7.6 LCS-Length, the arrow table, the reconstruction, and the two-row variant, with a timing and memory comparison. Compile with `g++ -std=c++17 -O2 -o lcs lcs.cpp`.....483
Listing 7.7 Counting calls against distinct subproblems for Merge Sort, naive Fibonacci and plain matrix-chain recursion — an empirical test for overlapping subproblems. Compile with `g++ -std=c++17 -O2 -o overlap overlap.cpp`.....487
Listing 7.8 Profitina Laundry: matching handwritten slips to the price list with LCS similarity, in $\Theta(\min(m,n))$ space. Compile with `g++ -std=c++17 -O2 -o laundry laundry.cpp`.....491
Listing 7.9 The Profitina Laundry bag-packing dynamic program of Exercise 7.18, with a greedy comparison. Compile with `g++ -std=c++17 -O2 -o bags bags.cpp`.....504
Listing 8.1 Problem 8.1 in full: `MAX-HEAPIFY` and `BUILD-MAX-HEAP` with a call-by-call trace, heapsort with the array after every extraction, and a measurement of the linear bound.....514
Listing 8.2 Problem 8.2 in full: a traced Lomuto partition, the complete recursion with per-call comparison counts, a generator of best-case inputs, and a measurement of the best, worst and random cases.....522
Listing 8.3 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.....527
Listing 8.4 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.....532
Listing 8.5 Problem 8.4 in full: the four-line recursive answer, the ascending walk beside it, the iterative answer with an explicit stack, and a trace confirming the $O(h)$ stack depth.....538
Listing 8.6 Problem 8.5 in full, with a node-visit counter that makes the difference between the permitted and the forbidden method measurable.....542
Listing 8.7 Problem 8.6 in full: the wasteful method, the one-pass sentinel method, the genuinely quadratic height-annotation routine, and a measured comparison over eleven tree shapes.....550
Listing 8.8 Problems 8.7 to 8.10 in one program: `insertHB`, `rebalanceForLeft`, `rebalanceForRight`, `deleteHB`, plus an AVL self-check, a sorted-order check and a stress test against the theoretical height bound.....567
Listing 8.9 Problems 8.11 to 8.15 in one program: the red-black validators, the imperative rotations with parent pointers, a set of checks, and a demonstration of the `int`-bound pitfall of §8.6.7.....575
Listing 8.10 Problem 8.16 in full: `MATRIX-CHAIN-ORDER` with a trace of every candidate split, `PRINT-OPTIMAL-PARENS`, and a count of the calls made by plain and memoized recursion.....585
Listing 8.11 Problem 8.17 in full: `LCS-LENGTH` with a trace of every cell, the combined table of values and arrows, and the walk back that produces one longest common subsequence.....594
Listing 8.12 Complete solutions to Problems 8.18 to 8.25, with a deliberate corruption of one stored field to show the validator of Problem 8.18 catching it, and a two-thousand-element stress test.....600
Listing 8.13 Profitina Laundry's pickup index, built entirely from the eight tree solutions of this chapter, with a self-audit that names the faulty record and a rebuild that survives a corrupted ordering.....608

Listing 8.14 Runnable solutions to exercises 4, 5, 7 and 8, with a `STRICT` switch that reproduces the `< 0` bug of exercise 8 on demand. 620
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`. 631
Listing 9.2 Activity selection in full: the recursive and iterative greedy selectors, the brute-force check, the four candidate rules with counterexamples for the three wrong ones, and a timing run. Compile with `g++ -std=c++17 -O2 -o activity activity.cpp` and run `./activity`. 648
Listing 9.3 The two knapsacks: greedy by density (optimal for the fractional problem, wrong for the 0-1 problem), the 0-1 optimum by dynamic programming, brute force as a check, and a stress test over 120 000 random instances. Compile with `g++ -std=c++17 -O2 -o knapsack knapsack.cpp` and run `./knapsack`. 660
Listing 9.4 A complete Huffman encoder and decoder: the traced construction on the frequencies of **Figure 9.7**, the codebook, $B(T)$ computed two independent ways, a real bit-packed round trip on ordinary text, the one-symbol edge case, and a timing run. Compile with `g++ -std=c++17 -O2 -o huffman huffman.cpp` and run `./huffman`. 675
Listing 9.5 Greedy(M, w) written once against an abstract independence oracle, then applied to the task-scheduling matroid (with a brute-force check and a 200 000-instance stress test) and to the graphic matroid, where it becomes Kruskal's algorithm. Compile with `g++ -std=c++17 -O2 -o matroid matroid.cpp` and run `./matroid`. 690
Listing 9.6 Profitina Laundry: activity selection on the dryer bookings (with a brute-force check), Huffman compression of the day's status log with a verified round trip, and the delivery van as a 0-1 knapsack where greed loses 10% of the revenue. Compile with `g++ -std=c++17 -O2 -o laundry laundry.cpp` and run `./laundry`. 696
Listing 9.7 The programs for Exercises 9.7, 9.10 and 9.15: shortest-job-first checked against all $n!$ orders, Huffman on Fibonacci frequencies, and weighted activity selection where the greedy rule fails. Compile with `g++ -std=c++17 -O2 -o exercises exercises.cpp` and run `./exercises` 714
Listing 10.1 The chapter's graph stored three ways at once, with all four basic queries answered from each representation. Compile with `g++ -std=c++17 -O2` 743
Listing 10.2 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 10.7** row by row. 754
Listing 10.3 Depth-first search with an explicit stack, printing the stack at every step. The `Frame` struct carries the resume position `next` — without it, returning to a vertex would restart its adjacency list and the search would loop. 770
Listing 10.4 Depth-first search, recursive, on the example of §10.8.4. It prints the trace, the timestamps, the parenthesization of §10.8.8, the forest, and the classification of every arc. 772
Listing 10.5 The second DFS example run in full, to check the timestamps and the parenthesization of **Figure 10.25** and **Figure 10.26**. 776
Listing 10.6 Connected components by repeated breadth-first search, on a graph shaped like **Figure 10.9**. Once `comp[]` is built, "are u and v connected?" becomes a single integer comparison. 782
Listing 10.7 Cycle detection for both undirected and directed graphs, tested on seven graphs including the chapter's own examples. The undirected test accepts any non-white neighbor other than the parent; the directed test insists on **gray**. 785
Listing 10.8 Bipartiteness testing by two-coloring during a BFS. On success it prints the two sides, which is the certificate; on failure it names the edge that could not be colored, which points at an odd cycle. 788
Listing 10.9 Topological sort of the dressing DAG by depth-first search, followed by a verification pass that checks every arc points forward in the produced order. 794
Listing 10.10 Timing BFS over an adjacency list against BFS over an adjacency matrix on the same graphs. A spanning tree is built first, so the graph is certainly connected and both searches must reach all V vertices — which the program checks before reporting any time. 797

Listing 10.11 The Profitina Laundry case study: breadth-first search on the pickup network of neighborhoods, and topological sort on the wash-process DAG, together with what happens when the process specification is made cyclic. 802
Listing 11.1 Topological sort by depth-first finishing times, on the dressing DAG. Compile with `g++ -std=c++17 -O2 -o toposort toposort.cpp`. 830
Listing 11.2 Kahn's in-degree topological sort, with the free cycle test. Compile with `g++ -std=c++17 -O2 -o kahn kahn.cpp`. 834
Listing 11.3 Prim–Jarník with a binary heap, tracing the queue at every step. Compile with `g++ -std=c++17 -O2 -o prim prim.cpp`. 855
Listing 11.4 A disjoint-set forest with union by rank and path compression. Compile with `g++ -std=c++17 -O2 -o dsu dsu.cpp`. 863
Listing 11.5 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`. 873
Listing 11.6 Prim and Kruskal measured against each other, including a thousand-graph randomized agreement test. Compile with `g++ -std=c++17 -O2 -o compare compare.cpp`. 876
Listing 11.7 The Profitina Laundry case study: a minimum spanning tree over ten branches, and a topological sort with a critical path over nine job stages. Compile with `g++ -std=c++17 -O2 -o laundry laundry.cpp`. 882
Listing 11.8 The broken directed test and the repaired undirected one, measured against seven graphs whose answers are known. Compile with `g++ -std=c++17 -O2 -o undirected undirected.cpp`. 890
Listing 12.1 Rute Laundry: breadth-first search and components, Kruskal with a disjoint-set forest, and Kahn's topological sort with the critical path. Compile with `g++ -std=c++17 -O2 -Wall -o router router.cpp` and run `./router`. 907
Listing 12.2 The evaluation harness: a cross-check of Kruskal against Prim, a postcondition check on BFS, and a timing run at five sizes with the elapsed time divided by the predicted cost. Compile with `g++ -std=c++17 -O2 -Wall -o routereval routereval.cpp` and run `./routereval`. 914
Listing 12.3 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`. 920
Listing 12.4 The matroid greedy algorithm for unit-time tasks with deadlines and refunds, with the independence counts printed at every step and a brute-force check over all 128 subsets. Compile with `g++ -std=c++17 -O2 -Wall -o tasks tasks.cpp`. 936
Listing 12.5 Worked Problems B to E, computed: activity selection under four rules against brute force; earliest deadline first against all 720 orders; Huffman with its tree, $B(T)$ both ways and a round trip; and the knapsack, fractional and 0-1. Compile with `g++ -std=c++17 -O2 -Wall -o greedy greedy.cpp` and run `./greedy`. 940
Listing 12.6 Depth-first search of an undirected graph, printing every discovery, every finish and every back edge in clock order, then the d/f table, both edge classes and the parenthesis structure; run from A and from E. Compile with `g++ -std=c++17 -O2 -Wall -o dfsu dfsu.cpp`. 955
Listing 12.7 Worked Problems G, H, J, K and L, computed: the BFS trace, the DFS timestamps and classification, the components, the two-coloring with its odd-cycle certificate, and both cycle tests. Compile with `g++ -std=c++17 -O2 -Wall -o search search.cpp`. 962
Listing 12.8 Worked Problems M to Q: the depth-first timestamps and classification, both topological sorts, the count of legal orders and the uniqueness test, the critical path, and the injected cycle. Compile with `g++ -std=c++17 -O2 -Wall -o dag dag.cpp`. 975
Listing 12.9 Prim's algorithm with `std::priority_queue` and lazy `DecreaseKey`, printing the heap after every step and counting the stale entries; run from a and from i. Compile with `g++ -std=c++17 -O2 -Wall -o primheap primheap.cpp`. 986
Listing 12.10 Worked Problems R, S, U, V and W: the Kruskal trace with the disjoint-set forest, the Prim trace with the key array, the cut and cycle properties, the union-by-rank sequence, and the uniqueness test run twice. Compile with `g++ -std=c++17 -O2 -Wall -o mst mst.cpp`. 995
Listing 12.11 The Profitina Laundry website audit: click depth and orphan pages by breadth-first search, near-duplicate clustering by a minimum spanning tree with the heavy edges cut, and the

remediation plan topologically sorted with its critical path. Compile with `g++ -std=c++17 -O2 -Wall -o laundry laundry.cpp`. 1005
Listing 12.12 The programs for Exercises 12.1, 12.3, 12.5, 12.7, 12.9 and 12.11. Compile with `g++ -std=c++17 -O2 -Wall -o exercises exercises.cpp` and run `./exercises`. 1014
Listing 13.1 Dijkstra and Bellman–Ford on a three-vertex graph with one negative edge and no cycle. Compile with `g++ -std=c++17 -O2 -o dijfail dijfail.cpp`. 1040
Listing 13.2 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`. 1045
Listing 13.3 A cross-check and a benchmark: heap Dijkstra, array Dijkstra and Bellman–Ford must agree on every non-negative graph, and the two Dijkstras are then timed on sparse ($m = 8n$) and dense ($m = n2/4$) inputs. 1052
Listing 13.4 Bellman–Ford with negative-cycle detection on the example graph, printing every pass. The second half of `main` 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 shows the detector firing. Compile with `g++ -std=c++17 -O2 -o bellman bellman.cpp`. 1061
Listing 13.5 `DAG-Shortest-Paths`: Kahn's topological sort followed by a single relaxation pass. The `if ((int)order.size() != n)` test is a free cycle detector — Kahn's algorithm cannot output every vertex if a cycle exists — so the program refuses to give an answer on a graph where its assumption fails. Compile with `g++ -std=c++17 -O2 -o dag dag.cpp`. 1069
Listing 13.6 Path reconstruction from π, iteratively and recursively, with two safety features: a cycle guard in `buildPath`, and a re-computation of each path's weight from the edge list to check it against d. The graph is the Dijkstra example graph plus an extra vertex q that nothing points at. 1073
Listing 13.7 The Profitina Laundry case study: Dijkstra on driving minutes, Bellman–Ford on net cost with one negative leg, and the negative-cycle detector firing when the rebate is raised too far. Compile with `g++ -std=c++17 -O2 -o laundry laundry.cpp`. 1078
Listing 13.8 Solution to Exercise 13.13: Bellman–Ford extended so that each vertex is labeled $+\infty$, a finite distance, or $-\infty$, according to whether it is unreachable, safely reachable, or reachable from a negative cycle. 1086
Listing 14.1 A stack with `Multipop`, instrumented. Compile with `g++ -std=c++17 -O2 -o multipop multipop.cpp`. 1097
Listing 14.2 The k-bit binary counter with a per-bit flip counter. Compile with `g++ -std=c++17 -O2 -o counter counter.cpp`. 1102
Listing 14.3 Exercise 17.1-3 checked numerically against the closed form and the bound. Compile with `g++ -std=c++17 -O2 -o ex1713 ex1713.cpp`. 1108
Listing 14.4 The accounting method instrumented: the credit balance printed after every operation, next to the quantity the proof claims it equals. Compile with `g++ -std=c++17 -O2 -o credit credit.cpp`. 1114
Listing 14.5 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`. 1120
Listing 14.6 The potential method instrumented on both the stack and the counter. Compile with `g++ -std=c++17 -O2 -o potential potential.cpp`. 1126
Listing 14.7 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`. 1131
Listing 14.8 `TABLE-INSERT` with doubling, instrumented with the potential function. Compile with `g++ -std=c++17 -O2 -o dyntable dyntable.cpp`. 1140
Listing 14.9 The two contraction policies compared on the thrashing sequence. Compile with `g++ -std=c++17 -O2 -o dyntable2 dyntable2.cpp`. 1144
Listing 14.10 Union–find with both tricks, every pointer counted. Compile with `g++ -std=c++17 -O2 -o dsu dsu.cpp`. 1152
Listing 14.11 Profitina Laundry: a dynamic table, a `Multipop` stack and a two-stack queue, all instrumented, over one working day. Compile with `g++ -std=c++17 -O2 -o laundry laundry.cpp`. 1157
Listing 14.12 A counter with `Increment` and `Reset`, both $O(1)$ amortized. Compile with `g++ -std=c++17 -O2 -o resetcounter resetcounter.cpp`. 1168
Listing 14.13 Three growth policies compared: doubling, tripling, and adding a fixed 100 slots. Compile with `g++ -std=c++17 -O2 -o growth growth.cpp`. 1172

Listing 14.14 Measuring what `std::vector<int>::push_back` really does. Compile with `g++ -std=c++17 -O2 -o vectorgrow vectorgrow.cpp`. 1178
Listing 15.1 Dijkstra audited against Floyd–Warshall on two graphs with negative edges and no negative cycles. Compile with `g++ -std=c++17 -O2 -o dijkstra_fails dijkstra_fails.cpp`. 1189
Listing 15.2 SLOW-ALL-PAIRS-SHORTEST-PATHS in full, printing every intermediate matrix. Compile with `g++ -std=c++17 -O2 -o slow_apsp slow_apsp.cpp`. 1201
Listing 15.3 FASTER-ALL-PAIRS-SHORTEST-PATHS, the repeated-squaring version. Compile with `g++ -std=c++17 -O2 -o faster_apsp faster_apsp.cpp`. 1208
Listing 15.4 The same triple loop with k outermost and with k innermost, on two graphs built to tell them apart. Compile with `g++ -std=c++17 -O2 -o ex_loop_order ex_loop_order.cpp`. 1215
Listing 15.5 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`. 1219
Listing 15.6 Negative-weight-cycle detection from the diagonal, on three graphs. Compile with `g++ -std=c++17 -O2 -o negcycle negcycle.cpp`. 1227
Listing 15.7 Five algorithms, one graph, one answer. Compile with `g++ -std=c++17 -O2 -o crosscheck crosscheck.cpp`. 1229
Listing 15.8 The transitive-closure algorithm in full. Compile with `g++ -std=c++17 -O2 -o transitive_closure transitive_closure.cpp`. 1235
Listing 15.9 The boolean triple loop, one byte per entry and one bit per entry, measured. Compile with `g++ -std=c++17 -O2 -o ex_bitset_tc ex_bitset_tc.cpp`. 1237
Listing 15.10 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`. 1244
Listing 15.11 Floyd–Warshall against $V \times$ Dijkstra at nine densities, $V = 600$ throughout. Compile with `g++ -std=c++17 -O2 -o crossover crossover.cpp`. 1249
Listing 15.12 Profitina Laundry: all-pairs driving times, eccentricity, the graph center, the diameter, and the route behind the worst run. Compile with `g++ -std=c++17 -O2 -o laundry laundry.cpp`. 1253
Listing 15.13 The same triple loop over three semirings: shortest paths, widest paths, and shortest-path counts. Compile with `g++ -std=c++17 -O2 -o ex_semirings ex_semirings.cpp`. 1264
Listing 16.1 An exhaustive refutation of statement (13): every graph on four labeled vertices is built, and the depth-first and breadth-first orders from vertex 0 are compared. 1279
Listing 16.2 The $O(n2)$ algorithm of Problem 16.2 in C++, printing the L and P tables. 1285
Listing 16.3 The $O(n\lg n)$ method for Problem 16.2, printing the `tail` array after every element. 1286
Listing 16.4 The experiment behind Problem 16.3: random binary trees of a given internal size, with both counts taken. Note that `externals` returns 1 for a null pointer — that is the definition of an external node, written as code. 1290
Listing 16.5 The complete approximation algorithm of Problem 16.4, with the triangle inequality verified, the minimum spanning tree built by Prim, the full and preorder walks printed, and the optimum found by brute force for comparison. 1297
Listing 16.6 Both algorithms of Problem 16.5 on the two graphs of Chapter 13, printing the estimate vector after each extraction and after each pass. 1306
Listing 16.7 Dijkstra's failure on a negative edge, and Bellman–Ford's negative-cycle detection, on the two smallest graphs that show each. 1308
Listing 16.8 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. 1312
Listing 16.9 Bellman–Ford pass by pass on the graphs of Problem 16.7, with the number of changes in each pass, the closing check, and — when the check fails — the negative cycle recovered from the predecessor array. 1316
Listing 16.10 Shortest paths in the DAG of Problem 16.8: a depth-first topological sort, then one relaxation per edge in topological order, with d/π printed after each vertex. 1321
Listing 16.11 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. 1326

Listing 16.12 The doubling dynamic table of Problem 16.10, with the actual cost, the running total against $3i$, the accounting bank, the potential $2\,num - size$ and the amortized cost of every insertion. 1330
Listing 16.13 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. 1333
Listing 16.14 Johnson's algorithm on the graph of Problem 16.12: Bellman–Ford from the auxiliary vertex, the reweighting with its arithmetic, and Dijkstra from every source, with the distances d under w' and the recovered distances D side by side. 1337
Listing 16.15 The longest path in a weighted DAG for Problem 16.14: Kahn's topological sort followed by one maximizing relaxation per edge. 1342
Listing 16.16 Floyd–Warshall and Johnson on one five-vertex graph, with all six intermediate matrices, the potentials and the reweighted edges printed. 1345
Listing 16.17 The 0/1 knapsack dynamic program of Problem 16.19, printing the whole c table and reconstructing the chosen set by walking it backward. 1350
Listing 16.18 The 2-approximation for vertex cover of Problem 16.20, with the matching it implicitly builds and the true optimum found by exhaustive search. 1352
Listing 16.19 The Profitina Laundry nightly planner: three questions from one data set, answered by a greedy algorithm, a minimum spanning tree with a preorder walk, and Dijkstra — one row of **Table 16.19** each. 1364
Listing 16.20 The Huffman code of Exercise 8, with the merge sequence, the codewords, and the comparison against a three-bit fixed-length code. 1373

List of Pseudocode

Pseudocode 2.1 Finding the largest of n numbers with one pass. 74
Pseudocode 2.2 Insertion Sort in the pseudocode used throughout this book, with 1-based indices. 76
Pseudocode 2.3 Linear search: examine the elements from left to right until q is found or the array runs out. 96
Pseudocode 2.4 Binary search on a sorted array: compare with the middle element and discard the half that cannot contain q. 99
Pseudocode 2.5 Summing the elements of an array. 114
Pseudocode 3.1 Recursive binary search for s in the sorted slice $A[p..r]$. One recursive call ($a = 1$) on a half-sized interval ($b = 2$), plus $\Theta(1)$ work per call, gives $T(n) = T(n/2) + \Theta(1)$. 166
Pseudocode 3.2 Merge Sort on the slice $A[p..r]$, with MERGE described in words; **Listing 3.8** gives the full code. 176
Pseudocode 4.1 Answering one LDCSUMAB query: a floating-point first guess for the block, two correcting loops, and the position inside the block. 199
Pseudocode 4.2 The TOHU algorithm: one evaluation of the closed form Equation (4.7) per query. 210
Pseudocode 4.3 Horner's rule: evaluating $a0 + a1x + \cdots + anxn$ from the highest coefficient down. 214
Pseudocode 4.4 Insertion Sort, as in Section 2.8.2, with indices running from 1. 217
Pseudocode 4.5 Binary search, as in Section 2.14.3, returning an index of q or NIL. 219
Pseudocode 4.6 A triple loop whose innermost line is to be counted exactly. 222
Pseudocode 4.7 The Laundry Tokens algorithm: one evaluation of Equation (4.14) per query. 239
Pseudocode 4.8 Fast doubling: the Fibonacci pair $(Fn, Fn + 1)$ modulo p in $\Theta(\log n)$ multiplications. 241
Pseudocode 4.9 The sort-then-search algorithm for the pair-sum problem, in $\Theta(n\lg n)$. 243
Pseudocode 5.1 `MAX-HEAPIFY`, also called `HEAPIFY`: let the value at node i sink until it dominates both of its children. 269
Pseudocode 5.2 `BUILD-MAX-HEAP`: repair every internal node, from the last one back up to the root. 276
Pseudocode 5.3 `HEAPSORT`: build a max-heap, then repeatedly move the root to the end of the shrinking heap and repair the root. 286
Pseudocode 5.4 `HEAP-MAXIMUM`: the largest key of a max-heap is always at the root. 293

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. 293
Pseudocode 5.6 `HEAP-INCREASE-KEY`: raise the key at position i, then float it up past every smaller parent. 294
Pseudocode 5.7 `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. 294
Pseudocode 5.8 `MAX-HEAP-INSERT`: append a slot holding $-\infty$, then raise it to the new key. 294
Pseudocode 5.9 Lomuto's `PARTITION`: one left-to-right scan that grows a region of elements no larger than the pivot. 301
Pseudocode 5.10 Hoare's `PARTITION`: two pointers close in from the ends and exchange misplaced pairs until they cross. 303
Pseudocode 5.11 `QUICKSORT` with Lomuto's partition: partition, then sort the two sides of the pivot recursively. 307
Pseudocode 5.12 `RANDOMIZED-PARTITION` and `RANDOMIZED-QUICKSORT`: swap a uniformly random element into the pivot slot, then partition as before. 320
Pseudocode 5.13 `QUICKSORT-TAIL`: recurse only on the smaller piece and loop on the larger, so that the stack depth stays $O(\lg n)$. 321
Pseudocode 6.1 The Left-Left repair: one right rotation about z, written as its four pointer assignments. 388
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. 389
Pseudocode 6.3 The Right-Right repair: one left rotation about z, the mirror image of **Pseudocode 6.1**. 390
Pseudocode 6.4 The Right-Left repair: a right rotation about y, then a left rotation about z. 390
Pseudocode 7.1 The straightforward recursion for Fn, transcribed directly from the definition. 438
Pseudocode 7.2 Memoized Fibonacci: the same recursion, consulting and filling a table of answers. 444
Pseudocode 7.3 Bottom-up Fibonacci: the table is filled from the smallest index upward. 444
Pseudocode 7.4 Fibonacci in constant space: a sliding window of two values replaces the table. 445
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. 459
Pseudocode 7.6 Print-Optimal-Parens: reading the optimal bracketing out of the c table. 462
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. 468
Pseudocode 7.8 The first line of **Lookup-Chain**: return a stored answer instead of recomputing it. 468
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"`. 478
Pseudocode 8.1 `MAX-HEAPIFY`: the value at node i sinks until neither child is larger than it. 511
Pseudocode 8.2 `BUILD-MAX-HEAP`: repair the internal nodes from the bottom up. 511
Pseudocode 8.3 Lomuto's `PARTITION`: one left-to-right sweep that grows a region of values $\leq x$. 518
Pseudocode 8.4 `QUICKSORT`: partition, then recurse on the two sides of the pivot. 519
Pseudocode 8.5 `MATRIX-CHAIN-ORDER`: every subchain is solved once, shortest first, and the winning split is recorded. 582
Pseudocode 8.6 `LCS-LENGTH`: the table of lengths and the table of arrows, filled row by row, with ties going up. 591
Pseudocode 8.7 `PRINT-LCS`: follow the arrows back; every diagonal arrow is one character of the answer. 591
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. 684
Pseudocode 10.1 Breadth-first search from a source s, in the notation of this book, with the line numbers of **Figure 10.15**. 750

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. 766
Pseudocode 10.3 Topological sort by depth-first search: the vertices in decreasing order of finishing time. 792
Pseudocode 10.4 The inner loop of BFS extended to count shortest paths: $cnt[v]$ accumulates the counts of all neighbors one level closer to the source. 816
Pseudocode 10.5 Testing whether an undirected graph on $n \geq 1$ vertices is a tree: one edge count and one breadth-first search. 817
Pseudocode 11.1 Topological sort by depth-first search, recording each vertex at the moment it finishes. 827
Pseudocode 11.2 Kahn's topological sort: repeatedly emit a vertex with no remaining predecessor; a shortfall at the end proves a cycle. 832
Pseudocode 11.3 The generic minimum-spanning-tree algorithm: grow a set A of edges one safe edge at a time. 847
Pseudocode 11.4 The generic algorithm with "safe" replaced by Corollary 11.6: choose a cut that respects A and take its light edge. 847
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. 849
Pseudocode 11.6 Making a singleton set, and union by rank: the root of smaller rank is hung under the root of larger rank. 861
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. 862
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. 866
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. 904
Pseudocode 12.2 The greedy change-making rule: repeatedly hand over the largest coin that does not exceed the amount still owed. 920
Pseudocode 12.3 The earliest-finish-time rule for activity selection, as one sweep over jobs already sorted by finish time. 925
Pseudocode 12.4 Huffman's algorithm: $n - 1$ merges of the two least frequent trees, each merge creating one internal node. 930
Pseudocode 12.5 The matroid greedy algorithm for unit-time tasks with deadlines and penalties: keep the costliest orders on time whenever independence allows. 936
Pseudocode 12.6 Kahn's algorithm: repeatedly emit a vertex with no remaining prerequisite, and remove its outgoing arcs. 972
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. 984
Pseudocode 13.1 Initialization shared by every single-source algorithm: all estimates infinite, no predecessors, the source at distance zero. 1033
Pseudocode 13.2 Relaxing the edge (u, v): one comparison, and two assignments when the route through u is strictly better. 1033
Pseudocode 13.3 Printing a shortest path from s to v by recursing on the predecessor array. 1073
Pseudocode 16.1 The $O(n2)$ dynamic program for the longest increasing subsequence, followed by the backtracking that recovers the subsequence itself. 1282
Pseudocode 16.2 Single-source shortest paths in a directed acyclic graph: one topological sort, then one relaxation per edge. 1319
Pseudocode 16.3 `INCREMENT` on a k-bit binary counter: clear the trailing 1 bits, then set the first 0 bit. 1323
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. 1327
Pseudocode 16.5 Longest paths from s in a weighted DAG: the DAG shortest-path algorithm with the comparison reversed. 1341
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. 1345

Pseudocode 16.7 The 0/1 knapsack dynamic program: n rows of $W + 1$ entries, each computed in constant time. 1350
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. 1354

List of Equations

(1.1) 13
(1.2) 19
(1.3) 19
(1.4) 23
(1.5) 26
(1.6) 30
(1.7) 31
(1.8) 34
(1.9) 38
(1.10) 39
(1.11) 39
(2.1) 74
(2.2) 83
(2.3) 85
(2.4) 86
(2.5) 86
(2.6) 87
(2.7) 88
(2.8) 91
(2.9) 95
(2.10) 97
(2.11) 101
(2.12) 104
(2.13) 120
(3.1) 126
(3.2) 130
(3.3) 131
(3.4) 132
(3.5) 138
(3.6) 138
(3.7) 141
(3.8) 144
(3.9) 145
(3.10) 146
(3.11) 150
(3.12) 160
(3.13) 160
(3.14) 163
(3.15) 165
(3.16) 177
(4.1) 197
(4.2) 198

(4.3) 198
(4.4) 205
(4.5) 207
(4.6) 208
(4.7) 208
(4.8) 214
(4.9) 223
(4.10)....224
(4.11)....230
(4.12)....232
(4.13)....233
(4.14)....239
(4.15)....240
(4.16)....241
(5.1) 262
(5.2) 262
(5.3) 263
(5.4) 265
(5.5) 274
(5.6) 282
(5.7) 283
(5.8) 283
(5.9) 302
(5.10)....310
(5.11)....310
(5.12)....311
(5.13)....313
(5.14)....317
(5.15)....317
(5.16)....320
(5.17)....325
(5.18)....325
(5.19)....336
(6.1) 352
(6.2) 354
(6.3) 376
(6.4) 376
(6.5) 378
(6.6) 379
(6.7) 380
(6.8) 380
(6.9) 380
(6.10)....383
(6.11)....410
(7.1) 434
(7.2) 437
(7.3) 440
(7.4) 441
(7.5) 441
(7.6) 446
(7.7) 452

(7.8) 455
(7.9) 455
(7.10) 458
(7.11) 458
(7.12) 464
(7.13) 473
(7.14) 477
(7.15) 491
(7.16) 503
(7.17) 504
(8.1) 514
(8.2) 514
(8.3) 521
(8.4) 521
(8.5) 521
(8.6) 527
(8.7) 527
(8.8) 531
(8.9) 547
(8.10) 549
(8.11) 549
(8.12) 554
(8.13) 554
(8.14) 556
(8.15) 558
(8.16) 560
(8.17) 564
(8.18) 582
(8.19) 585
(8.20) 591
(9.1) 635
(9.2) 638
(9.3) 638
(9.4) 640
(9.5) 641
(9.6) 656
(9.7) 657
(9.8) 668
(9.9) 672
(9.10) 673
(9.11) 684
(9.12) 686
(9.13) 686
(10.1) 725
(10.2) 727
(10.3) 733
(10.4) 759
(10.5) 760
(10.6) 760
(10.7) 761
(10.8) 762

(10.9) 762
(10.10) 769
(10.11) 775
(10.12) 779
(11.1) 839
(11.2) 842
(11.3) 842
(11.4) 845
(11.5) 854
(11.6) 854
(11.7) 861
(11.8) 862
(11.9) 872
(11.10) 873
(12.1) 903
(12.2) 903
(12.3) 904
(12.4) 917
(12.5) 920
(12.6) 926
(12.7) 927
(12.8) 929
(12.9) 931
(12.10) 932
(12.11) 934
(12.12) 935
(12.13) 949
(12.14) 959
(12.15) 971
(12.16) 973
(12.17) 983
(12.18) 986
(12.19) 993
(13.1) 1023
(13.2) 1024
(13.3) 1027
(13.4) 1027
(13.5) 1028
(13.6) 1029
(13.7) 1036
(13.8) 1049
(13.9) 1050
(13.10) 1051
(13.11) 1057
(13.12) 1065
(13.13) 1066
(14.1) 1092
(14.2) 1092
(14.3) 1101
(14.4) 1106
(14.5) 1106

(14.6) 1107
(14.7) 1107
(14.8) 1112
(14.9) 1119
(14.10) 1121
(14.11) 1123
(14.12) 1123
(14.13) 1136
(14.14) 1138
(14.15) 1148
(14.16) 1150
(14.17) 1150
(14.18) 1174
(15.1) 1185
(15.2) 1186
(15.3) 1194
(15.4) 1195
(15.5) 1196
(15.6) 1196
(15.7) 1213
(15.8) 1213
(15.9) 1218
(15.10) 1218
(15.11) 1234
(15.12) 1234
(15.13) 1240
(15.14) 1242
(16.1) 1269
(16.2) 1282
(16.3) 1289
(16.4) 1293
(16.5) 1295
(16.6) 1322
(16.7) 1323
(16.8) 1324
(16.9) 1327
(16.10) 1332
(16.11) 1335
(16.12) 1337
(16.13) 1350

Preface

There is a particular kind of silence in a lecture hall when a student has understood something. Not the silence of confusion, which is heavy, but the quiet that arrives a half-second before someone says “oh — of course.” Every page of this book was written trying to earn that half-second.

This is a book about designing algorithms and proving things about them. It grew out of the Design and Analysis of Algorithms course (EF234405) taught in the Department of Informatics at Institut Teknologi Sepuluh Nopember (ITS), Surabaya, and it has been rebuilt so that it stands entirely on its own: you do not need the course, a lecturer, or any other document to learn from it. Its sixteen chapters follow the rhythm of a sixteen-week semester, and the illustrations, worked examples and traces in it are the ones that have been tested, year after year, on real students in a real room.

Who this book is for

You are assumed to know nothing. Not “nothing about algorithms” — nothing. If you have never proved anything, never written a line of C++, never seen a logarithm used in anger, you are exactly the reader this book was written for. Every symbol is introduced in words before it is used. Every formula is derived rather than announced. Every example is worked by hand, one step to a line, with the arithmetic shown, because the arithmetic is where understanding either happens or quietly fails to.

That principle has a practical consequence: this book is long. A shorter book would be a book that assumed you already knew the hard parts, and a book that assumes you already know the hard parts is of no use to the person who does not. Where the choice was between brevity and a reader who finishes able to implement the material, the reader won.

What is in it

Chapters 1 to 3 build the foundations: what an algorithm is, how to prove one correct, the RAM model, asymptotic notation, recurrences and the Master Theorem. Chapters 5 to 7 cover sorting, balanced search trees, divide and conquer, and dynamic programming. Chapters 9 to 11 turn to greedy algorithms and to graphs — searching them, and spanning them. Chapters 13 to 15 treat shortest paths, amortized analysis, and the all-pairs problem.

Chapters 4, 8, 12 and 16 are different in kind. Each is an **Evaluation and Discussion** chapter that closes one quarter of the book. Nothing new is introduced there; instead, the material of the three chapters before it is put to work on complete problems, solved in full: every problem restated in plain words, every step of the reasoning written out, the arithmetic shown, the tempting wrong turns named and taken apart, a second method given wherever one exists, and a complete program that was compiled and run. Each problem ends with a “check yourself” variant, so that you can test whether the method has become yours.

Every program in this book is complete, compiles with a standard C++17 compiler, and was run before its output was printed here. No output appears on these pages that a machine did not

produce. The same programs are collected, chapter by chapter, in the accompanying `Code` folder, each with a README that explains how to compile and run it on Windows, macOS and Linux.

How to read it

Slowly, and with a pen. Work each example on paper before reading the worked solution. When a proof appears, cover the next line and try to supply it. When a program appears, type it in and run it, then break it deliberately and see what the breakage looks like — that is often where the real learning is.

The exercises at the end of each chapter all have complete solutions. Use them the way you would use the answers at the back of a mathematics text: after your own attempt, not instead of it.

Numbered figures, tables, listings, pseudocode and equations are all live links: in the electronic edition, click "Figure 5.3" or "[4]" anywhere in the text and you are taken straight to it. The contents and the lists of figures, tables, listings, pseudocode and equations at the front work the same way.

About the authors

Irfan Subakti describes himself in one line — *Father | Educator | Entrepreneur | Bridging Tech & Nature: CS Lecturer × Crypto/Digital Business × AI/ML Specialist × Fruit Farmer | Storyteller* — and every part of that line has found its way into this book. He has taught in the Department of Informatics at ITS since 1998. He supervises and examines undergraduate theses, teaches MLOps for the IBM SkillsBuild program, and builds products: he founded Profitina, an AI-visibility platform whose engineering appears in the "Profitina in Practice" section of every chapter. Away from the keyboard he tends the family fruit and durian orchard, which is where he learned that the best explanations, like the best durian trees, take years of patient pruning. The storyteller in him is the reason this book prefers a story to a list whenever a story will do.

Rully Soelaiman has taught algorithms in the same department at ITS for many years; generations of students know him simply as Pak Rully, the best lecturer in Competitive Programming. A great deal of the material in this book — problems, worked examples, illustrations and explanations refined over many years in the classroom — began as his, and he generously gave it to this course and to this book.

Acknowledgments

My deepest debt is to **Rully Soelaiman, S.Kom., M.Kom.** — my teacher, my patron, and above all my dear friend. Pak Rully to everyone else; Masbro Rully to me. So much of this book began with him: the problems he chose, the examples he worked out, the illustrations he drew, the explanations he shaped over years of teaching and then gave freely so that others could learn. It is only right that his name now stands beside mine on the cover. Much of what a reader will find clear here is clear because he made it so; whatever is still unclear is mine. Thank you, Masbro — for the material, for the standard you set, for the patience with which you taught me, and for a friendship that has outlasted every syllabus we have ever taught together. This book is, in no small part, yours.

The material owes its shape, too, to the students who worked through its earlier forms and asked the questions that revealed where an explanation was thin. It owes its standard to Cormen,

Leiserson, Rivest and Stein, whose *Introduction to Algorithms* remains the measure against which any book of this kind should be held. Where this book differs from theirs, it differs by going slower.

The worked applications throughout draw on **Profitina**, an AI-visibility platform built in Surabaya, and on **Profitina Laundry**, the teaching case study that accompanies this book. The first shows where the material lands in a real product; the second is small enough that you can run all of it yourself.

Surabaya, September 2026

Irfan Subakti · yifana@gmail.com

Rully Soelaiman · rully130270@gmail.com

Profitina

profitina.com

Chapter 1

Foundations, Framework, and First Problems

By the end of this chapter you will know what an algorithm is and how it differs from the program that implements it; you will be able to state precisely what it means for an algorithm to be *correct* and what it means for it to be *efficient*; you will know the five great families of algorithm design and be able to recognize which family a new problem belongs to; and you will have solved six real competition problems from beginning to end — from the first reading of the statement to a C++ program that compiles, runs, and produces the judge's expected answer — using a five-phase method that you will use for the rest of this book and, if this book does its job, for the rest of your working life.

1.1 Where We Begin

This is a book about learning to **think computationally**. That phrase is used loosely in a great many places, so let us pin it down. To think computationally about a problem means four things, and we will practice all four in every chapter:

1. **Model the problem algorithmically.** Somebody hands you a description of a situation in ordinary human language — a delivery van, a pile of laundry, a sequence of digits on a screen. You must turn it into a precise mathematical object: exactly what comes in, exactly what must go out, and exactly which inputs are possible.
2. **Design an algorithm.** Choose a strategy — and there are only a handful of genuinely different strategies, which is the good news — and lay out the steps that turn the input into the output.
3. **Analyze correctness and complexity.** *Prove* that your steps always give the right answer, for every input that the problem allows, not merely for the three examples you happened to try. Then compute how much time and how much memory the steps consume as the input grows.
4. **Implement it cleanly.** Translate the design into working C++ using the right data structures.

Notice the order. Understand, then design, then analyze, then code. **The code is the last step, not the first.** Many students, and a surprising number of working programmers, jump straight to the keyboard. They open an editor, type `int main() {`, and start guessing. Sometimes they get away with it, on easy problems. On hard problems they produce hundreds of lines of code that is wrong in a way they cannot find, because they never decided what "right" was supposed to mean.

The thinking that happens before the code is written is exactly what separates a good engineer from a great one. This book is mostly about that thinking.

The book is paced like a one-semester university course in the design and analysis of algorithms: sixteen chapters, one for each week, in which every fourth chapter (Chapters 4, 8, 12 and 16) stops to evaluate and discuss the three chapters before it. Section 1.15 lays out the whole road map.

A word about the reader this book assumes. It assumes nothing. If you have never written a line of C++, never seen the symbol $\lfloor x \rfloor$, never heard the word *logarithm*, you are the reader we had in

mind. Every symbol is introduced in words before it is used. Every formula is derived rather than announced. Every example is worked by hand, one line at a time, with the arithmetic shown. When an idea is hard, it is stated once in the language of mathematics and then again in the language of a market stall. You may read this book with a pencil, a sheet of paper, and a computer with a C++ compiler, and nothing else.

1.2 Algorithms and Data Structures — the Two Pillars

1.2.1 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 the name of the ninth-century Persian scholar Muhammad ibn Musa al-Khwarizmi, whose book on calculation with Hindu-Arabic numerals reached medieval Europe under the Latinized title *Algoritmi de numero Indorum*. The idea is older than the word: any reliable recipe for getting an answer is an algorithm.

A recipe is in fact the best first analogy. Consider making *nasi goreng*. You have ingredients — cold rice, an egg, shallots, *kecap manis*. That is the **input**. You follow steps in an order: heat the wok, fry the shallots, add the rice, and so on. At the end you have a plate of fried rice. That is the **output**.

For a procedure to count as an algorithm in the technical sense, it must have five properties, in the classical formulation given by Knuth [1]. Each of these will matter later, so let us take them one at a time with the recipe in mind.

Property 1 — a clearly defined input. The algorithm must say exactly what it receives. "A sequence of n whole numbers, each between 1 and 1000" is a clearly defined input. "Some numbers" is not, because it does not tell you whether negative numbers can occur, or whether the list can be empty.

Property 2 — a clearly defined output. The algorithm must say exactly what it produces, and what makes that production *right*. "Print the smallest number" is clearly defined. "Print a good number" is not.

Property 3 — definiteness. Every step must be precisely specified, with no ambiguity. "Add salt to taste" is fine in a cookbook, because a human cook can taste. It is not an algorithmic step, because a machine cannot. "Add 3 grams of salt" is an algorithmic step.

Property 4 — finiteness. The procedure must eventually stop on every valid input. A set of steps that loops forever is not an algorithm; it is a bug. This is not a trivial requirement. Consider the instruction "start at n; if n is even divide it by two, if n is odd replace it with $3n + 1$; repeat until you reach 1." Nobody on earth knows whether that stops for every starting value — it is the famous Collatz conjecture, open since 1937. We therefore cannot call it an algorithm for anything.

Property 5 — effectiveness. Every step must be basic enough that it can actually be carried out. "Compute the largest prime number" is not effective, because there is no largest prime. "Decide whether an arbitrary program halts" is not effective either, by a celebrated result of Alan Turing.

> **EXAMPLE.** Is this an algorithm? > *Input:* a list of n numbers. > *Step 1:* look at the first number and remember it. > *Step 2:* look at each remaining number in turn; if it is bigger than what you remember, remember it instead. > *Step 3:* when no numbers remain, announce what you remember.

Input: defined. Output: defined — it is the largest number in the list. Definiteness: "bigger than" is unambiguous for numbers. Finiteness: the list has n items and we look at each once, so we stop after n steps. Effectiveness: comparing two numbers is a basic operation. Yes — this is an algorithm, and it is the very first one we will write in C++.

1.2.2 A program is an implementation of an algorithm

This distinction is small but it is the source of a great deal of confusion, so we shall be blunt about it.

DEFINITION. A **program** is the expression of an algorithm in a particular programming language, so that a particular machine can run it.

The algorithm is the *idea*. The program is the *realization* of the idea. The same algorithm — "remember the biggest so far" — can be written in C++, in Python, in Java, on paper, or explained out loud to a friend. It is the same algorithm every time. Only the program changes.

This matters for three practical reasons.

First, **you can be wrong in two different places.** Your algorithm can be wrong (the idea does not solve the problem), or your program can be wrong (the idea is fine but you typed `<` where you meant `<=`). These are different diseases and they need different medicines. If you have proved the algorithm correct on paper and the program still fails, you have a coding bug and you should read the code. If you have not proved anything, you do not know where to look.

Second, you can compare algorithms without writing any code. We will spend a whole chapter learning to say that one algorithm is a million times faster than another without ever compiling either.

Third, the algorithm is the durable part. C++ will be replaced one day. Binary search will not.

1.2.3 What a data structure is

DEFINITION. A **data structure** is a way of organizing, storing and managing data so that it can be accessed and modified efficiently.

Here is an analogy worth keeping in mind:

An algorithm is like a cooking recipe — the steps. A data structure is like the kitchen equipment — the pots, the pans, the oven. The same recipe with better equipment produces the dish faster.

The choice of data structure can be the entire difference between a program that finishes in milliseconds and a program that never finishes at all. In Section 1.11 you will meet a problem where the input can be as large as 10^{18}. Stored in an ordinary array, the answer would need more memory than exists on Earth. Stored in a `std::map`, it needs about a thousand entries and runs instantly. The algorithm is the same in both cases. Only the container changed.

These are the structures you will meet in this book. You are not expected to know any of them yet; each gets its own treatment when it is needed.

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

Table 1.1 is a map of the territory, not a list to memorize now. You will meet each entry when a problem forces you to want it, which is the only way anyone ever really learns a data structure.

1.2.4 The goals we are aiming at

Figure 1.1 summarizes what we are trying to achieve, and it divides the goals into two groups.

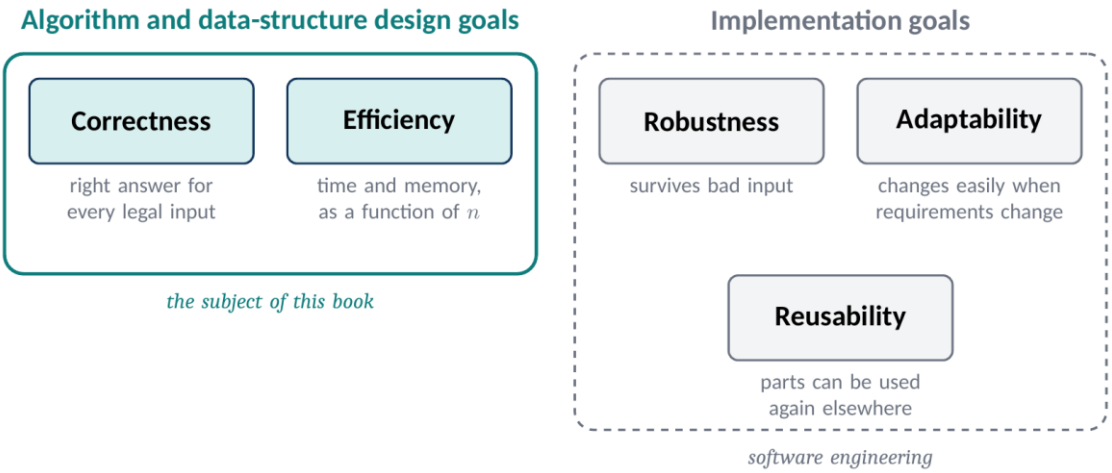


Figure 1.1 The goals of data structure and algorithm design (correctness, efficiency) set beside the goals of implementation (robustness, adaptability, reusability).

The left-hand pair — **correctness** and **efficiency** — belong to the design of the algorithm itself, and they are the subject of this book. The right-hand trio — **robustness** (the program survives bad input), **adaptability** (it can be changed when requirements change) and **reusability** (parts of it can be used again elsewhere) — belong to software engineering. They are real and they matter, but an algorithm that is fast, reusable, robust and *wrong* is worthless. So we begin on the left.

1.3 The Two Goals of Algorithm Design

When we design an algorithm there are exactly two things we must guarantee. Not three, not one. Two.

1.3.1 Goal number 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.

Read that definition again and notice the word *every*. Not "the examples in the statement". Not "the cases I thought of". Not "small inputs". Every input that the constraints permit.

This is a stronger demand than most beginners realize, and it is the reason we cannot establish correctness by testing. Suppose a problem says $1 \leq n \leq 1000$ and $1 \leq a_i \leq 1000$. Then the number of possible inputs is astronomically large — for $n = 1000$ alone it is 1000^{1000}, a number with three thousand and one digits. You could test one input per nanosecond until the sun burns out and not scratch it. Testing can show the presence of bugs; it can never show their absence. Only a proof can do that.

There are four standard ways to prove an algorithm correct, and you will use all four in this book.

Direct proof. Show, by straightforward reasoning from the definitions, that the algorithm's logic always leads to the required answer. Most of the six problems in this chapter are proved directly: we show that the answer must be at least some value, then we *construct* an arrangement achieving exactly that value, and the two halves pin the answer down.

Proof by contradiction. Assume the algorithm produces a wrong answer on some input, and derive from that assumption something impossible. Since the impossible cannot happen, the assumption must be false, so the algorithm is right. We will use this for the *Journey* problem in Section 1.7: we assume a tour shorter than $2 \times$ (range) exists, and show that it must fail to visit one of the cities.

Loop invariant. Identify a property that is true before a loop begins, stays true across every single pass of the loop, and — combined with the reason the loop ended — gives exactly the result we wanted. This is the workhorse technique for any algorithm built around a loop, and it deserves a section of its own. See Section 1.3.3.

Induction. Prove the statement for the smallest case, then prove that if it holds for a case of size k it must hold for size $k + 1$. Since it holds at the bottom, and each rung of the ladder leads to the next, it holds everywhere. We use induction in Section 1.10 to prove a formula about a sequence that doubles in length forever.

1.3.2 A problem stated precisely: sorting

Before we can prove anything about a sorting algorithm we must say exactly what sorting *is*. This is Phase 1 of the framework applied to the most-studied problem in computer science, and **Figure 1.2** pictures it.

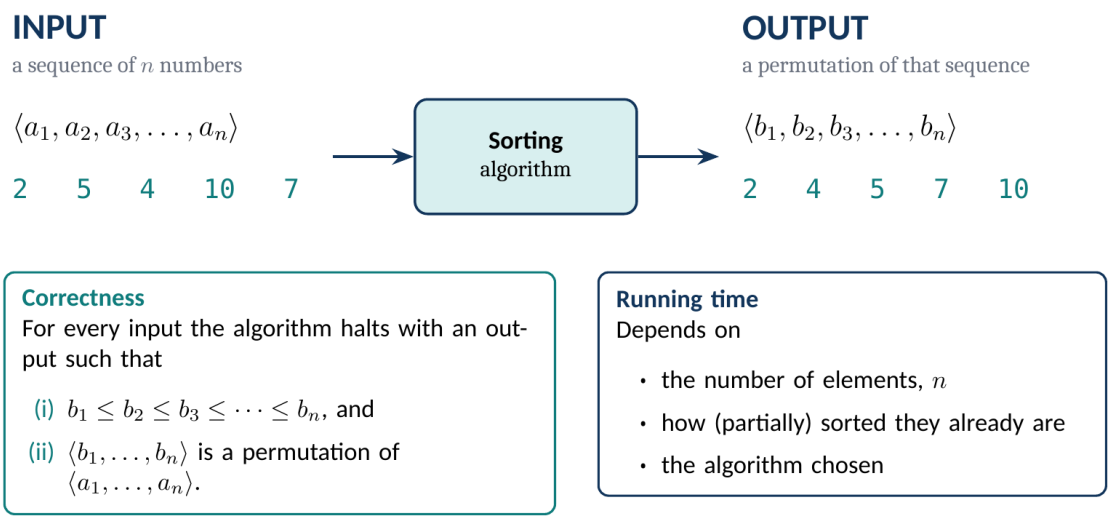


Figure 1.2 The sorting problem: a sequence goes in, a permutation of that sequence comes out, and correctness means two separate conditions must both hold.

> **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 $a_1, a_2, \ldots, a_n$.

Let us unpack every symbol, because this is the first formal problem statement in the book.

The letter n stands for *how many numbers there are*. The notation $a_1, a_2, \ldots, a_n$ means: the first number is called a_1 (read "a sub one"), the second is called a_2, and so on up to the last, which is called a_n. The little number written below and to the right is called a **subscript** or **index**; it is a label, not a multiplication. The symbol $\le$ means "is less than or equal to". The word **permutation** means a rearrangement: the same collection of numbers, possibly in a different order, with nothing added and nothing thrown away.

Condition (i) says the output is in non-decreasing order. Condition (ii) says the output contains exactly the same numbers as the input.

Both conditions are needed, and here is why. Take the input

$$a = [\,2,\ 5,\ 4,\ 10,\ 7\,].$$

An algorithm that outputs [2,4,5,7,10] is correct: the sequence is in order, and it uses exactly the numbers 2,4,5,7,10, which are exactly the numbers we were given.

An algorithm that outputs [1,2,3,4,5] satisfies condition (i) beautifully — it is perfectly sorted — and fails condition (ii) completely. The numbers 1 and 3 were never in the input; the numbers 7 and 10 have vanished. It is **wrong**. A checker that asks only "is the output sorted?" will happily accept this algorithm. This is why the problem statement, not the test, defines correctness.

An algorithm that outputs [2,5,4,10,7] unchanged satisfies (ii) and fails (i). Also wrong.

1.3.3 Insertion sort, and the loop invariant that proves it

Here is a sorting algorithm you already know, even if you have never seen it written down: it is how nearly everybody sorts a hand of playing cards. You hold the cards you have already arranged in your left hand. You pick up the next card with your right hand. You move it leftwards past every card that is bigger than it, until you meet one that is not, and you drop it into the gap. Repeat until the table is empty.

That is **insertion sort**. **Listing 1.1** is the whole of it in C++.

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

```
#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;
}
```

Compiled with `g++ -std=c++17 -O2` and run, **Listing 1.1** prints:

```
2 4 5 7 10
```

which is exactly the output shown on the right of **Figure 1.2**. But a correct answer on one input proves nothing, as we argued above. Let us prove it properly.

> **DEFINITION.** A **loop invariant** is a statement about the program's variables that is (a) **true before the loop runs at all** — *initialization*; (b) **still true at the end of each pass, given that it was true at the start of that pass** — *maintenance*; and (c) **useful when the loop finally stops** — *termination*.

The invariant is the bridge between "I ran the loop a few times and it looked fine" and "I know what this loop does for every input." The idea goes back to Floyd and Hoare in the 1960s and was worked into a complete discipline of programming by Gries [2], whose book is still the clearest treatment of where invariants come from. Here is the invariant for insertion sort.

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

Initialization. Before the first pass, $j = 1$. The sub-array $a[0 \dots 0]$ is a single element. A single element is trivially in sorted order — there is no pair of elements that could be out of order — and it is trivially the original occupant of position 0, because nothing has moved yet. The invariant holds.

Maintenance. Assume the invariant holds at the start of a pass, so $a[0 \dots j-1]$ is sorted and contains the right elements. Inside the body we save `key = a[j]`. The `while` loop then copies $a[i]$ into $a[i+1]$ for each i walking leftwards, as long as $a[i] > \texttt{key}$. Two things are true about this shifting:

- Nothing is lost. The value that gets overwritten at position $i+1$ has already been copied one step further right (or, on the very first shift, it is `key` itself, which we saved in a variable). So the multiset of values in $a[0 \dots j]$ is unchanged.

- The loop stops at the first position i where $a[i] \leq \texttt{key}$, or at $i = -1$ if it ran off the left end. Writing `key` into $a[i+1]$ therefore puts it after every element that is $\leq$ it and before every element that is $>$ it, and since the elements to its left and right were already sorted among themselves, the whole of $a[0 \ldots j]$ is now sorted.

After the body, j increases by one, so the invariant's claim about $a[0 \ldots j-1]$ is exactly the claim we just established about $a[0 \ldots j]$. The invariant is maintained.

Termination. The loop stops when $j = n$. Substituting $j = n$ into the invariant gives: $a[0 \ldots n-1]$ contains exactly the original elements and is in sorted order. That is precisely conditions (ii) and (i) of the sorting problem. The algorithm is correct. ■

Notice that we never mentioned the particular numbers 2,5,4,10,7. The proof covers every input of every size — which is what a proof is for.

To see the invariant with your own eyes, **Listing 1.2** prints the array after each pass with a bar marking the boundary between the sorted part and the untouched part.

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

```
#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;
}
```

Its actual output is:

```
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
```

Read the left of each bar. `2`, then `2 5`, then `2 4 5`, then `2 4 5 10`, then the whole array. Sorted every time, and always containing exactly the elements that started in those positions. That is the invariant, made visible.

1.3.4 Goal number two: efficiency

An algorithm must also use as little time and as little memory as it reasonably can. Two algorithms can both be perfectly correct and still be separated by a gulf that no faster computer can bridge.

> **SORTING ONE MILLION NUMBERS.** Bubble sort performs roughly n^2 operations. With $n = 10^6$ that is $(10^6)^2 = 10^{12}$ operations — about an hour on a modern machine. Merge sort performs roughly $n\log_2 n$ operations. With $n = 10^6$ that is about 2×10^7 operations — about twenty milliseconds. Both are correct. One is roughly fifty thousand times faster.

We measure efficiency with **Big-O notation**, which we will define rigorously in Chapter 3. For now, treat $O(f(n))$ as a statement about *how the running time grows as the input grows*, ignoring constant factors and small terms. If an algorithm is $O(n^2)$ and you double n, expect the time to quadruple. If it is $O(n)$ and you double n, expect the time to double. If it is $O(\log n)$ and you double n, expect the time to go up by a single step.

Table 1.2 lists the growth rates you will meet, with the operation count each implies at $n = 1000$.

Table 1.2 The common growth rates, and what each one costs at $n = 1000$.

Notation	Name	Operations at n=1000	How it feels
$O(1)$	Constant	1	Instant
$O(\log n)$	Logarithmic	about 10	Instant
$O(n)$	Linear	1,000	Fast
$O(n\log n)$	Linearithmic	about 10,000	Fast
$O(n^2)$	Quadratic	1,000,000	Slow once n grows
$O(n^3)$	Cubic	10^9	Very slow
$O(2^n)$	Exponential	about 10^{301}	Impossible

Two entries in **Table 1.2** deserve a word. The logarithm $\log_2 n$ is *the number of times you can halve n before reaching* 1. Halving 1000 gives 500,250,125,62,31,15,7,3,1 — nine or ten steps. That is why $O(\log n)$ is written as "about 10" in the table. And 2^n at $n = 1000$ is 2^{1000}, which is roughly 10^{301}: a number with three hundred and two digits. The observable universe contains something like 10^{80} atoms. An exponential algorithm at $n = 1000$ is not slow; it is impossible in the strict physical sense.

1.3.5 The rule of thumb that decides everything

Here is the single most practically useful fact in this chapter.

> **KEY INSIGHT — THE OPERATION BUDGET.** A modern computer performs roughly 10^8 to 10^9 simple operations per second. If your algorithm, on the largest input the constraints allow, would perform substantially more than about 10^8 operations, it will very probably exceed the time limit.

A "simple operation" here means one addition, one comparison, one array access — the kind of thing that turns into a handful of machine instructions. Operations involving a `map` or a division are several times more expensive; operations involving reading from disk are thousands of times more expensive. The figure 10^8 is deliberately crude. It is crude and it is enough.

Why does this rule matter so much? Because it turns the constraints printed in a problem statement into a *design decision made before you write any code*. Look at the largest n the problem

allows, decide what complexity fits in the budget, and only then start designing. **Table 1.3** does the arithmetic for you.

Table 1.3 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 1.3**. When a problem allows n up to 10^{18} — and Section 1.11 contains exactly such a problem — you cannot even *read* n items, let alone process them. Any algorithm that touches each of n things is dead before it starts. The constraint is telling you, loudly, that the intended solution never looks at the input one item at a time. It must exploit structure. Beginners see 10^{18} and panic. Experienced problem solvers see 10^{18} and smile, because the constraint has just eliminated ninety percent of the search space of possible designs.

1.3.6 The two goals work together

Correctness without efficiency is useless: your program is right but it never finishes, so the judge reports *Time Limit Exceeded* and you score zero. Efficiency without correctness is equally useless: your program finishes instantly and prints the wrong number, so the judge reports *Wrong Answer* and you score zero. There is no partial credit for being half right at speed.

You need both, simultaneously, and the rest of this book is about how to get both.

1.4 The Five Design Paradigms

There is no single recipe for designing algorithms. If there were, this book would be one page long. What there is instead is a small number of *paradigms* — general strategies, each of which applies to a broad family of problems and each of which has recognizable symptoms. Learning to recognize the symptoms is most of the skill.

There are five in this book.

1.4.1 Incremental, or brute force

The idea. Build the solution one piece at a time. Process the input elements one by one, maintaining some running state, and when the input runs out the running state *is* the answer.

Insertion sort is incremental: it maintains a sorted prefix and grows it by one card per pass. So is the simplest algorithm of all, scanning for a maximum, shown in **Listing 1.3**.

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

```
#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;
}
```

Its actual output is:

```
maximum = 9
```

The loop invariant is `best` *holds the largest value among* $a[0 \ldots i-1]$. Before the loop, $i = 1$ and `best` $= a[0]$, which is indeed the largest of a one-element range. Each pass compares $a[i]$ with the champion and keeps the winner, so the invariant survives. When the loop stops, $i = n$ and `best` is the largest of $a[0 \ldots n-1]$: the whole array. Two of this chapter's six problems — *Columns* and *Journey* — are solved by exactly this shape of loop.

The term **brute force** is used when the incremental process simply tries every possibility. Brute force is often the right first thought, because it is usually obviously correct, and having something obviously correct to compare against is worth a great deal when you are debugging something clever.

1.4.2 Greedy

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

Greed is seductive and frequently wrong. If you are standing on a hillside in fog and always walk uphill, you will reach *a* summit, but not necessarily the highest one. A greedy algorithm is correct only when the problem possesses what is called the **greedy-choice property**: taking the locally best option never destroys the possibility of a globally best solution. When a problem has it, greedy algorithms are wonderfully short and fast. When it does not, they are wonderfully short, fast and wrong.

Proving the greedy-choice property is therefore not optional, and Chapter 9 is devoted to it. In this chapter, three of the six problems are greedy — *Columns*, *Journey*, and the digit construction in *Minimum and Maximum Numbers* — and in each case we will prove the greedy choice is safe rather than assume it.

The classic example you will meet later is Kruskal's algorithm for a minimum spanning tree: repeatedly add the cheapest remaining edge that does not create a cycle. It is provably optimal, and the proof is not obvious.

1.4.3 Divide and conquer

The idea. Break the problem into smaller sub-problems *of the same kind*, solve those recursively, then combine their answers into an answer for the whole.

Merge sort is the standard illustration, and **Listing 1.4** is a complete implementation.

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

```
#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;
}
```

Its actual output is:

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

The three characteristic steps are visible in the code. *Divide*: compute `mid` and cut the range in two. *Conquer*: two recursive calls. *Combine*: the merging loops. Chapter 3 shows how to compute the running time of such an algorithm from a recurrence, and Chapter 7 develops the paradigm in full.

1.4.4 Dynamic programming

The idea. Like divide and conquer, break the problem into sub-problems — but here the sub-problems **overlap**: the same sub-problem is needed many times. Solve each one once, store the answer, and look it up thereafter.

The canonical demonstration is the Fibonacci sequence, defined by $F(0) = 0$, $F(1) = 1$, and $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$. Written as a direct recursion, computing $F(30)$ recomputes $F(28)$ twice, $F(27)$ three times, $F(26)$ five times, and so on: the number of calls itself grows like a Fibonacci number, which is to say exponentially. **Listing 1.5** measures the damage and then repairs it.

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

```
#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;
}
```

Its actual output is:

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

Two million six hundred ninety-two thousand five hundred thirty-seven calls to compute a number that memoization reaches with twenty-nine stored values. And `fibMemo(90)` returns instantly, while `fibNaive(90)` would take longer than the age of the universe. Storing answers instead of recomputing them is called **memoization**, and Section 1.11 applies it to the last problem of this chapter.

1.4.5 Mathematical and number-theoretic

The idea. Some problems dissolve entirely once you see the right piece of mathematics. No loop over the data, no recursion — just a formula.

The classical example is the **greatest common divisor**. The greatest common divisor of two positive whole numbers a and b, written $\gcd(a, b)$, is the largest number that divides both of them exactly. The obvious method is to test every candidate from 1 upward, which takes about $\min(a, b)$ steps. Euclid's method, from around 300 BC, instead uses the identity

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

where $a \bmod b$ means the remainder when a is divided by b. Why is the identity true? Write $a = qb + r$, where q is the whole-number quotient and $r = a \bmod b$. Any number d that divides both a and b also divides $a - qb = r$; and any number that divides both b and r also divides $qb + r =$

a. So the pairs (a, b) and (b, r) have exactly the same common divisors, and therefore the same *greatest* common divisor.

Each application of Equation (1.1) replaces the pair by a strictly smaller pair, because the remainder r is always less than b. It shrinks fast, too: after any two consecutive steps the larger number has fallen to less than half of what it was (if $b \le a/2$ then $r < b \le a/2$; if $b > a/2$ then $r = a - b < a/2$). A number can be halved only about $\log_2$ of itself times before it reaches zero, so only $O(\log\min(a, b))$ steps are ever needed. **Listing 1.6** puts the two methods side by side.

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

```
#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;
}
```

Its actual output is:

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

Let us trace Euclid's method on gcd(1071,462), one line per step:

$$\gcd(1071,462): \quad 1071 \bmod 462 = 147 \;\Rightarrow\; \gcd(462,147)$$

$$\gcd(462,147): \quad 462 \bmod 147 = 21 \;\Rightarrow\; \gcd(147,21)$$

$$\gcd(147,21): \quad 147 \bmod 21 = 0 \;\Rightarrow\; \gcd(21,0)$$

$$\gcd(21,0) = 21.$$

Three steps, where exhaustive search would have taken four hundred and sixty-two. Section 1.8 contains a problem that is *entirely* an application of this one identity — and the last two lines of the output above, $\gcd(30,8) = 2$ and $\gcd(15,12) = 3$, are the two numbers we shall need there.

1.4.6 The five paradigms in one table

Table 1.4 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*

Table 1.4 is worth returning to whenever you are 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.

1.5 The Five-Phase Problem-Solving Framework

Every problem in this book — every problem in the four evaluation chapters (Chapters 4, 8, 12 and 16), every problem you will ever be handed at work — is solved by the same five phases, in the same order. This is the single most important thing in the chapter. Learn it here and you will use it for twenty years.

Table 1.5 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

Table 1.5 names the five phases and, more usefully, names what each one must *produce*. A phase you cannot show an artifact for is a phase you skipped. If you have no hand-worked example you have not done Phase 1; if you have no complexity figure you have not done Phase 4.

1.5.1 Phase 1 — Understand

Read the problem. Then read it again. Then, without looking at it, write down in your own words what it is asking. If you cannot, you have not understood it, and everything you build from here will be built on sand.

Concretely, Phase 1 produces four things:

- **The input.** Exactly what arrives, in exactly what order, in exactly what format.
- **The output.** Exactly what must be printed, in exactly what format, including whether there is a trailing newline and whether numbers are separated by spaces or by line breaks.

- **The constraints.** The smallest and largest values of everything. These are not decoration. As **Table 1.3** showed, the constraints decide the design.
- **A small example, worked by hand.** Not by computer. By hand, on paper, with a pencil. This is the step most often skipped and it is the step that most often finds the misunderstanding.

> **WARNING.** The sample input and sample output in the problem statement are part of the specification, not an afterthought. If your hand-worked example disagrees with the sample, you have misread the problem. Do not proceed. Find out why.

1.5.2 Phase 2 — Abstract

Strip the story away. The cities and the Cossack and the laundry are decoration; underneath there is a mathematical object. Ask: what am I really being asked for? What properties must the answer have? What quantity is being minimized or maximized, and over what set?

This is where the real thinking happens, and it is where a solution goes from "a loop over everything" to "one line of arithmetic". In Section 1.7 a traveling-salesman-sounding problem collapses into the expression $2(x_{\max} - x_{\min})$. In Section 1.10 a self-doubling sequence collapses into counting the 1-bits of a number. Neither collapse is visible until the story is removed.

A useful habit in Phase 2: try to prove a **lower bound** (the answer cannot possibly be smaller than L) and then try to **construct** something achieving L. If both succeed, the answer is exactly L and you are done. Two of the six problems below are solved precisely this way.

1.5.3 Phase 3 — Design

Now, and only now, choose a paradigm from **Table 1.4** and write the algorithm down. Write it in numbered steps, in English or pseudocode. Name the data structures. Do not write C++ yet.

A design written in English can be checked by a friend, argued about, and thrown away cheaply. A design written in C++ acquires an emotional weight — you have invested in it — and you will defend it long after you should have abandoned it.

1.5.4 Phase 4 — Analyze

Two questions, both compulsory.

Is it correct? Produce a proof, using one of the four techniques of Section 1.3.1. If the algorithm is a loop, state the invariant. If it is greedy, prove the greedy choice is safe. If it is recursive, use induction. "It worked on the sample" is not an answer.

Is it fast enough? Count the operations as a function of the input size, express it in Big-O, put the largest allowed input into the formula, and compare the result against the 10^8 budget. If it does not fit, go back to Phase 2 — not to Phase 5. You cannot optimize your way out of the wrong asymptotic class by writing tighter loops.

Also state the **space** complexity, and check it against the memory limit, which is typically 128 or 256 megabytes. An `int` occupies 4 bytes, so an array of 10^8 `int`s needs 400 megabytes and will not fit.

1.5.5 Phase 5 — Implement

Write the code. It should be a direct, clean translation of Phase 3, and it should be short.

> **THINK.** If your code looks complicated, your design is probably not finished. Complexity in the implementation is nearly always a symptom of an unresolved question in Phase 2. Go back and simplify the idea; do not try to tidy the code.

Most students skip straight to Phase 5. Resist this. The first four phases are exactly what make Phase 5 trivial, and you will notice, in the six problems that follow, that the finished programs are between eight and twenty lines long. That is not because the problems are easy. It is because the thinking was done first.

1.6 Problem 1 — Columns (E-Olymp 10281)

Paradigm: greedy / direct computation.

1.6.1 Phase 1 — Understand

Figure 1.3 is the problem exactly as it appears on the E-Olymp judge.

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

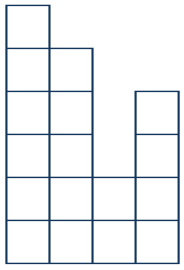


Figure 1.3 *Columns*, E-Olymp problem 10281, restated from the judge: the statement, the picture of four columns of cubes, and the input format.

In words: you are given n columns of cubes standing side by side on the ground. The i-th column is a_i cubes tall. You must paint every cube some color, subject to two rules, and you must use as few distinct colors as possible.

The input. The first line holds one integer n, the number of columns, with $1 \leq n \leq 1000$. The second line holds n integers $a_1, a_2, \ldots, a_n$, the heights, with $1 \leq a_i \leq 1000$.

The output. One number: the smallest number of colors that suffices.

The two rules.

1. Every **column** must have all its cubes in different colors.
2. Every **substring** must have all its cubes in different colors.

What is a substring? This is the one piece of vocabulary the problem invents, and misreading it is the commonest way to lose this problem. A *substring* is a horizontal run of cubes sitting next to each other at the same height **with no gap between them**. If a column is too short to reach that height, it leaves a hole, and the hole splits the row into two separate substrings that are allowed to repeat each other's colors.

> **HAND-WORKED EXAMPLE — heights** [6,5,2,4]**.** Let us draw the shape row by row, from the top down. Write `[X]` for a cube and a blank for empty air.

Row 6: `[X]` — only column 1 is this tall, so one cube alone Row 5: `[X][X]` — columns 1 and 2; a substring of length 2 Row 4: `[X][X] [X]` — column 3 has height 2, so there is a **gap**; two substrings, of lengths 2 and 1 Row 3: `[X][X] [X]` — same gap at column 3 Row 2: `[X][X][X][X]` — every column reaches height 2; one substring of length 4 Row 1: `[X][X][X][X]` — every column reaches height 1; one substring of length 4

That hand-drawn picture, which takes half a minute with a pencil, contains the entire solution. Look at it and two facts jump out. The tallest column has 6 cubes stacked vertically, and rule 1 says they must all differ, so we need at least 6 colors. And the bottom row is an unbroken substring of all 4 columns, so rule 2 says we need at least 4 colors. Six is the larger demand.

Figure 1.4 shows the judge's own sample: input `4` then `6 5 2 4`, with answer `6`, together with a legal painting.

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

```
4
6 5 2 4
```

Answer #1

```
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.

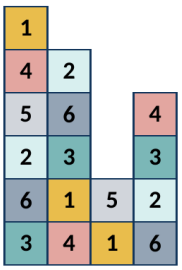


Figure 1.4 The sample test of *Columns* and one valid painting with six colors. Note the third row from the bottom: it contains the color 3 twice, which is legal because an empty space separates the two cubes.

Look carefully at the third row from the bottom in **Figure 1.4**. It reads 2, 3, gap, 3. The color 3 appears twice in the same row — and this is **legal**, because the gap left by the short third column splits that row into two independent substrings. This is the judge confirming, in a picture, exactly what the word *substring* means.

1.6.2 Phase 2 — Abstract

Strip away the cubes and the paint. We are looking for one number C, the number of colors. Two constraints push C upward, and we must find the smallest C that satisfies both.

Step 1 — the vertical constraint. Take the tallest column; call its height $H = \max(a_1, a_2, \dots, a_n)$. It contains H cubes, all in one column, and rule 1 forces them all to have different colors. You cannot give H objects different colors using fewer than H colors. Therefore

$$C \geq H = \max_i a_i.$$

Step 2 — the horizontal constraint. Consider row 1, the bottom row. Every column has height at least 1 — the constraints guarantee $a_i \geq 1$ — so every one of the n columns has a cube in row 1. There are no gaps. The whole bottom row is therefore a single substring of length n, and rule 2 forces all n of those cubes to have different colors. Therefore

$$C \geq n.$$

Notice we only needed the *bottom* row. Rows further up can only be shorter or broken, so they never demand more than n.

Step 3 — combine. Both demands must be met at once, so

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

Equation (1.2) is a **lower bound**: the answer cannot be smaller than this. To finish, we must show the bound is achievable — that $\max(n, \max_i a_i)$ colors really are enough.

Step 4 — prove sufficiency by construction. Let $C = \max(n, \max_i a_i)$. Number the columns $j = 0,1,\ldots,n-1$ from the left and the rows $r = 0,1,2,\ldots$ from the bottom. Paint the cube in column j, row r with color

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

In Equation (1.3), $x \bmod C$ means the remainder when x is divided by C; it is always one of $0,1,\ldots,C-1$. Adding 1 turns it into a color name from 1 to C.

Why every column is safe. Fix a column j. As r runs over $0,1,\ldots,a_j - 1$, the quantity $j + r$ takes a_j consecutive values. Consecutive integers have different remainders modulo C as long as there are at most C of them — and $a_j \leq \max_i a_i \leq C$. So all the colors in the column differ.

Why every substring is safe. Fix a row r and a gap-free run of columns $j, j+1, \ldots, k$. As j runs across the run, $j + r$ again takes consecutive values, at most n of them, and $n \leq C$. So all the colors in the substring differ.

Both rules hold, using exactly C colors. The lower bound is achieved, so the answer is exactly C.

> **KEY INSIGHT.** answer $= \max(n,\ \max(a_1, a_2, \ldots, a_n))$.

If you doubt the construction — and you should doubt constructions until you have checked them — **Listing 1.7** applies it to the sample heights and tests *every* column and *every* gap-free run by brute force.

Listing 1.7 A brute-force verifier for the coloring constructed in the proof of *Columns*.

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

int a[] = {6, 5, 2, 4};                 // the heights from the sample
const int n = 4;

int color(int j, int r, int C) {       // column j (0-based), row r (0-based)
    return ((j + r) % C) + 1;           // the construction from the proof
}

int main() {
    int C = n;
    for (int j = 0; j < n; j++) C = max(C, a[j]);
    printf("C = %d\n", C);

    bool bad = false;
    for (int j = 0; j < n; j++)         // every column: all colors distinct?
        for (int r = 0; r < a[j]; r++)
            for (int s = r + 1; s < a[j]; s++)
                if (color(j, r, C) == color(j, s, C)) bad = true;

    for (int r = 0; r < C; r++)         // every maximal gap-free run in row r
        for (int j = 0; j < n; j++)
            for (int k = j + 1; k < n; k++) {
                bool run = true;        // is j..k one unbroken run at height r?
                for (int t = j; t <= k; t++) if (a[t] <= r) run = false;
                if (run && color(j, r, C) == color(k, r, C)) bad = true;
            }

    printf(bad ? "FAILED: a clash exists\n" : "VERIFIED: no clash anywhere\n");
    return 0;
}
```

Its actual output is:

```
C = 6
VERIFIED: no clash anywhere
```

1.6.3 Phase 3 — Design

1. Read n.
2. Set $ans \leftarrow n$.
3. For each of the n heights a_i read from the input: set $ans \leftarrow \max(ans, a_i)$.
4. Print ans.

No array is needed. The heights are consumed one at a time and thrown away; only the running maximum survives. This is the incremental paradigm of Section 1.4.1 in its purest form.

1.6.4 Phase 4 — Analyze

Correctness. Proved above: $\max(n, \max_i a_i)$ is both a lower bound (Steps 1–3) and achievable (Step 4). The loop invariant for step 3 is *after processing i heights, $ans = \max(n, a_1, \ldots, a_i)$*, which holds initially with $i = 0$ since $ans = n$, is maintained by the `max`, and at $i = n$ delivers the answer.

Time. One pass over n numbers, constant work per number: $O(n)$. With $n \leq 1000$ that is a thousand operations, against a budget of 10^8. We are using one hundred-thousandth of the time limit.

Space. Three integers: $O(1)$.

1.6.5 Phase 5 — Implement

Listing 1.8 The complete solution to *Columns*, E-Olymp 10281.

```
#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;
}
```

On the judge's sample input

```
4
6 5 2 4
```

Listing 1.8 actually printed:

```
6
```

matching the expected answer in **Figure 1.4**. As a second check, on the input `5` followed by `1 1 1 1 1` — five columns, each a single cube — the program printed `5`, which is right: the tallest column demands only 1 color, but the unbroken bottom row of five demands 5.

1.7 Problem 2 — Journey (E-Olymp 10280)

Paradigm: greedy / mathematical observation.

1.7.1 Phase 1 — Understand

Journey (E-Olymp 10280)

Time limit 1 s Memory limit 128 MB

Statement

Cossack Vus is going on a journey. In Potokolandia n cities lie on a straight line, numbered 1 to n; city i has coordinate x_i, and the distance between cities i and j is $|x_i - x_j|$. Vus must visit every city at least once and finish in the city where he started; the starting city and the route are his to choose. Find the minimum length of such a route.

Input

The first line contains one integer n $(1 \le n \le 100)$. The second line contains n integers $x_1, x_2, \ldots, x_n$ $(1 \le x_i \le 1000)$.

Figure 1.5 *Journey*, E-Olymp problem 10280. Cossack Vus must visit every city in Potokolandia and come home again.

As **Figure 1.5** states it: in Potokolandia there are n cities on a **straight line**, numbered 1 to n. City i sits at coordinate x_i. The distance between cities i and j is $|x_i - x_j|$ — the vertical bars mean

absolute value, that is, the difference with any minus sign thrown away, so that distance is never negative.

Cossack Vus wants to visit every city at least once and finish in the city where he started. He may choose *both* the starting city *and* the route. We must report the minimum total distance he travels.

The input. First line: one integer n with $1 \leq n \leq 100$. Second line: n integers $x_1, \ldots, x_n$ with $1 \leq x_i \leq 1000$.

The output. One integer, the minimum route length, as **Figure 1.6** shows.

Journey (E-Olymp 10280)

Time limit 1 s Memory limit 128 MB

Output

Print one integer: the minimum length of the route of Cossack Vus.

Sample 1

Input #1	Answer #1
2 1 4	6

Sample 2

Input #2	Answer #2
3 1 3 1	4

Figure 1.6 The two sample tests of *Journey*. Two cities at 1 and 4 give 6; three cities at 1,3,1 give 4.

HAND-WORKED EXAMPLE — sample 1. Two cities, at coordinates 1 and 4. Start at city 1 (coordinate 1). Walk right to coordinate 4: that is $|4 - 1| = 3$ units. Now visit city 2 — done. Walk back left to coordinate 1: another 3 units. Total $3 + 3 = 6$. The expected answer is 6.

HAND-WORKED EXAMPLE — sample 2. Three cities, at coordinates 1, 3 and 1. Note that two cities share the coordinate 1; nothing forbids this. Start at coordinate 1. City 3 is also at coordinate 1, so it is visited for free. Walk right to coordinate 3: 2 units. Walk back: 2 units. Total 4. The expected answer is 4.

1.7.2 Phase 2 — Abstract

Step 1 — reduce to one-dimensional geometry. Forget the names of the cities, forget Cossack Vus. We have n points on a number line. We need a closed walk — a walk that ends where it began — passing through all of them, of minimum total length.

This *sounds* like the traveling salesman problem, which is one of the hardest problems in computer science. On a line, it is not. Geometry rescues us.

Step 2 — the only two moves. On a line you can move left or move right. That is all. So any route is a sequence of leftward and rightward moves, and the total distance is the sum of their lengths.

Step 3 — find the lower bound. Let $x_{\min}$ be the smallest coordinate and $x_{\max}$ the largest. Define the **range** $R = x_{\max} - x_{\min}$.

The route must at some moment stand on $x_{\min}$ (a city is there, and it must be visited) and at some moment stand on $x_{\max}$. Because the route is *closed* — it ends where it began — it is really a loop, and a loop has the same length whichever of its points we start reading it from. So let us start

reading it at a moment when it stands on $x_{\min}$. Reading forward from there, the loop splits into two pieces:

- **Out.** The walk eventually arrives at $x_{\max}$. Getting from $x_{\min}$ to $x_{\max}$ costs at least R units, because on a line no walk between two points is shorter than the distance between them.
- **Back.** The walk then carries on until it has gone all the way round the loop and stands on $x_{\min}$ again. Getting from $x_{\max}$ back to $x_{\min}$ costs at least R once more.

The two pieces do not overlap, so every valid closed route costs at least $R + R = 2R$.

Put as a **proof by contradiction**: suppose some closed route cost *less* than $2R$. Cut it into the "out" and "back" pieces just described. Their lengths add up to less than $2R$, so at least one of them is shorter than R. But each piece joins $x_{\min}$ to $x_{\max}$, two points exactly R apart — and no walk between them can be shorter than R. That is impossible, so no such route exists.

Step 4 — achieve the bound. Start at $s = x_{\min}$. Walk right all the way to $x_{\max}$, passing through every city on the way, since every x_i lies between the two extremes. That costs R. Walk back left to $x_{\min}$. That costs another R. Total $2R$, and every city has been visited. The bound is achieved.

Step 5 — the starting point does not matter. Suppose you start at any s with $x_{\min} \le s \le x_{\max}$. Walk left to $x_{\min}$: cost $s - x_{\min}$. Walk right all the way to $x_{\max}$: cost $x_{\max} - x_{\min}$. Walk back left to s: cost $x_{\max} - s$. Adding:

$$(s - x_{\min}) + (x_{\max} - x_{\min}) + (x_{\max} - s).$$

The two terms containing s are $+s$ and $-s$; they cancel. What is left is

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

The start cancels out algebraically. Every starting city gives the same optimum, which is why the problem can let you choose it.

> **KEY INSIGHT.** answer $= 2 \times (\max_i x_i - \min_i x_i)$. The number of cities is irrelevant; only the two extremes matter.

Check Equation (1.4) against the samples. Sample 1: $x_{\max} = 4$, $x_{\min} = 1$, so $2(4 - 1) = 6$. Sample 2: $x_{\max} = 3$, $x_{\min} = 1$, so $2(3 - 1) = 4$. Both agree with **Figure 1.6**.

1.7.3 Phase 3 — Design

1. Read n.
2. Initialize $d_{\min}$ to a value larger than any legal coordinate and $d_{\max}$ to a value smaller than any legal coordinate.
3. For each coordinate read: update $d_{\min}$ and $d_{\max}$.
4. Print $2 \times (d_{\max} - d_{\min})$.

The constraints say $1 \le x_i \le 1000$, so starting from $d_{\min} = 2000$ and $d_{\max} = 0$ is safe: the first coordinate read will replace both.

1.7.4 Phase 4 — Analyze

Correctness. Proved above by a lower bound plus a matching construction. The invariant of the loop is *after reading i coordinates,* $d_{\min}$ *and* $d_{\max}$ *are the smallest and largest among them*.

Time. $O(n)$; with $n \leq 100$, one hundred operations. **Space.** $O(1)$ — the coordinates are never stored.

1.7.5 Phase 5 — Implement

Listing 1.9 The complete solution to *Journey*, E-Olymp 10280.

```
#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;
}
```

The expression `dmin < dist ? dmin : dist` is the **conditional operator**: it evaluates the test before the `?`, and yields the value before the `:` if the test is true, otherwise the value after it. So this line keeps whichever of `dmin` and `dist` is smaller. The next line, with the branches swapped, keeps whichever is larger.

On sample input 1 (`2` then `1 4`) **Listing 1.9** actually printed `6`; on sample input 2 (`3` then `1 3 1`) it actually printed `4`. Both match **Figure 1.6**.

1.8 Problem 3 — Unique Numbers (SPOJ DCEPC901)

Paradigm: number theory (greatest common divisor).

1.8.1 Phase 1 — Understand

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 $(i+x) \bmod n$; `1 x` replaces each number i by $(i \times x) \bmod n$. 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 1.7 *Unique Numbers*, SPOJ problem DCEPC901, with its two operation types and the crucial note that operations compose.

Figure 1.7 sets the scene. We begin with the whole numbers

$$0,1,2,\ldots,n-1,$$

all n of them, all different. Then k operations are applied, one after another. Each operation is given as two integers p and x:

- If $p = 0$: **replace each number i by $(i + x) \bmod n$.**
- If $p = 1$: **replace each number i by $(i \times x) \bmod n$.**

The note in bold on the judge page matters: *each operation is done on the numbers we get after the previous operation, not on the original*. The operations compose.

After each operation we must report how many **distinct** values remain. Numbers can collide, and when two of them become equal the count drops.

Constraints

$1 \le n \le 10^8$
$1 \le k \le 20$
$0 \le p \le 1$
$0 \le x \le 10^8$

Sample

Input	Answer
30 3 1 8 0 3 1 12	15 15 5

Figure 1.8 The constraints and the sample test of *Unique Numbers*: n up to 10^8, at most 20 operations.

Figure 1.8 gives the constraints: $1 \le n \le 10^8$, $1 \le k \le 20$, $0 \le p \le 1$, $0 \le x \le 10^8$.

Stop and read $n \le 10^8$ against **Table 1.3**. If we actually built the list of n numbers we would need 10^8 integers — four hundred megabytes, over the memory limit — and each of 20 operations would touch all of them, giving 2×10^9 operations, twenty times over the time budget. The constraints are telling us that the intended solution never materializes the list at all.

The sample in **Figure 1.8** is $n = 30$ with $k = 3$ operations `1 8`, `0 3`, `1 12`, and the expected output is 15, 15, 5.

1.8.2 Phase 2 — Abstract

Step 1 — addition never changes the count. Consider the map $g(i) = (i + x) \bmod n$. Take two different values $a \neq b$, both in $\{0, \ldots, n-1\}$. Could they collide? If $(a + x) \equiv (b + x) \pmod{n}$, subtract x from both sides: $a \equiv b \pmod{n}$. Since a and b both lie in $\{0, \ldots, n-1\}$, congruence modulo n forces $a = b$, contradicting $a \neq b$. So different inputs give different outputs: g is **injective** (one-to-one). A one-to-one map from a finite set to itself is also onto, so g is a **bijection** — it merely shuffles the numbers around.

Therefore: **an addition operation never changes the number of distinct values.** This is why the second output in the sample repeats the first.

Step 2 — multiplication shrinks the universe. Now consider $f(i) = (i \times x) \bmod n$ for $i = 0,1,\ldots,n-1$. Which values can come out?

Let $g = \gcd(n, x)$. We claim the set of outputs is exactly the multiples of g inside $\{0,\ldots,n-1\}$, namely

$$\{\,0,\ g,\ 2g,\ \ldots,\ n-g\,\},$$

which has n/g elements.

Every output is a multiple of g. The number ix is a multiple of x, and x is a multiple of g, so ix is a multiple of g. Reducing modulo n subtracts some multiple of n, and n is also a multiple of g. A multiple of g minus a multiple of g is a multiple of g. Done.

Every multiple of g is achieved. Bézout's identity says there are whole numbers u and v with $ux + vn = g$. Reducing modulo n kills the vn term, so $ux \equiv g \pmod{n}$: taking $i = u \bmod n$ produces the output g. And then $i = (2u) \bmod n$ produces $2g$, and so on. Every multiple of g appears.

So a multiplication by x collapses the universe from n values to $n/\gcd(n, x)$ values, evenly spaced g apart. In symbols, the number of distinct outputs is

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

where the vertical bars around a set mean "the number of elements in it".

Figure 1.9 illustrates this collapse, computed for $n = 15$ and four different multipliers.

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 1.9 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.

Read **Figure 1.9** one block at a time, checking each block against Equation (1.5).

- $x = 3$: the outputs are 0,3,6,9,12,0,3,6,9,12, … — only five different values, repeating three times. And $\gcd(15,3) = 3$, so $15/3 = 5$. Correct.
- $x = 5$: the outputs are 0,5,10,0,5,10, … — three different values. $\gcd(15,5) = 5$, so $15/5 = 3$. Correct.
- $x = 2$: the outputs are 0,2,4,6,8,10,12,14,1,3,5,7,9,11,13 — all fifteen values, merely shuffled. $\gcd(15,2) = 1$, so $15/1 = 15$. Correct.
- $x = 4$: the outputs are 0,4,8,12,1,5,9,13,2,6,10,14,3,7,11 — again all fifteen. $\gcd(15,4) = 1$. Correct.

Four independent confirmations of the formula, drawn by hand before any code was written. This is Phase 1 and Phase 2 working together exactly as they should.

Step 3 — chain the operations. After a multiplication, the surviving values are $0, g, 2g, \ldots$ — a set of size $n' = n/g$ which behaves, after dividing every element by g, precisely like the set $\{0,1,\ldots,n'-1\}$. So we may simply *replace* n *by* n' and carry on. Additions leave n alone. That is the whole algorithm: we never track the numbers, only how many of them there are.

> **KEY INSIGHT.** Keep n as a running variable. On a multiplication by x, set $n \leftarrow n/\gcd(n,x)$. On an addition, do nothing. Print n after every operation.

> **HAND-WORKED EXAMPLE — the judge's sample.** Start with $n = 30$.
>
> Operation 1 is `1 8`: a multiplication by 8. gcd(30,8): 30 mod 8 = 6, so gcd(8,6); 8 mod 6 = 2, so gcd(6,2); 6 mod 2 = 0, so the answer is 2. $n \leftarrow 30/2 = 15$. **Print 15.**
>
> Operation 2 is `0 3`: an addition. Additions are bijections. n stays 15. **Print 15.**
>
> Operation 3 is `1 12`: a multiplication by 12. gcd(15,12): 15 mod 12 = 3, so gcd(12,3); 12 mod 3 = 0, so the answer is 3. $n \leftarrow 15/3 = 5$. **Print 5.**
>
> The three printed numbers are 15,15,5 — exactly the expected output in **Figure 1.8**.

1.8.3 Phase 3 — Design

1. Read n and k.
2. Repeat k times: read p and x; if p is odd (that is, $p = 1$) set $n \leftarrow n/\gcd(n,x)$; print n.

1.8.4 Phase 4 — Analyze

Correctness. Established in Steps 1–3: addition is a bijection so the count is unchanged; multiplication by x maps the universe onto the multiples of $\gcd(n,x)$, of which there are $n/\gcd(n,x)$; and the surviving set is order-isomorphic to a fresh $\{0,\ldots,n'-1\}$, so the argument repeats.

Time. One gcd per operation. Euclid's algorithm takes $O(\log n)$ steps, so the whole program is $O(k\log n)$. With $k \le 20$ and $n \le 10^8$ that is at most a few hundred operations — nothing.

Space. $O(1)$. Four integers. Compare this with the four hundred megabytes the naive approach would have needed.

1.8.5 Phase 5 — Implement

Listing 1.10 The complete solution to *Unique Numbers*, SPOJ DCEPC901.

```
#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;
}
```

Three details of **Listing 1.10** deserve comment.

`__gcd` is a function supplied by the GNU C++ standard library in `<algorithm>`; it implements Euclid's algorithm from Section 1.4.5. In C++17 and later you may equally write `std::gcd` from `<numeric>`.

`p & 1` is the **bitwise and** of p with 1, which is 1 when p is odd and 0 when p is even. Since p is only ever 0 or 1, this is a terse way of writing `p == 1`.

`while (k--)` reads the current value of `k` as the loop condition and *then* decreases it, so the body runs exactly k times and `k` ends at -1.

Given the judge's sample input

```
30
3
1 8
0 3
1 12
```

Listing 1.10 actually printed

```
15
15
5
```

which is exactly the expected output. Note that the program reads `n` and `k` with a single `scanf("%d %d", ...)` even though the statement puts them on separate lines; `scanf` skips over any whitespace, newlines included, so this is safe.

> **A TRAP WORTH KNOWING.** The constraints allow $x = 0$. Multiplying every number by zero makes them all zero, so exactly one distinct value should remain. Does the formula survive? By convention $\gcd(n, 0) = n$, so $n \leftarrow n/n = 1$. It does survive. Running **Listing 1.10** on the input `30`, `2`, `1 0`, `0 5` actually printed `1` then `1`, which is right — and once everything has collapsed to a single value, no later operation can ever increase the count again.

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

Paradigm: greedy construction.

1.9.1 Phase 1 — Understand

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 $1 \leq N \leq 18$ and $1 \leq K \leq 10$. 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

```
2
1 1
3 3
```

Answer

```
1 9
102 987
```

Figure 1.10 *Minimum and maximum numbers*, SPOJ problem NEWLANG, with its constraints and sample test.

From **Figure 1.10**: given N and K, find the smallest and the largest positive integers that have **exactly** N **digits** and use **exactly** K **distinct digits** from $\{0,1,\ldots,9\}$. Leading zeros are not allowed.

The input. First line: T, the number of test cases (about 200). Each of the next T lines contains N and K, with $1 \leq N \leq 18$ and $1 \leq K \leq 10$. The input is guaranteed to be such that an answer always exists.

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

The sample. $T = 2$; the cases are `1 1` and `3 3`; the expected output is `1 9` and `102 987`.

Let us make sure we understand "exactly K distinct digits". The number 102 is written with the three characters 1, 0, 2; the set of digits used is $\{0,1,2\}$, which has three elements, so 102 has exactly 3 distinct digits. The number 111 has one distinct digit. The number 121 has two distinct digits $\{1,2\}$ even though it has three digit positions. Distinctness is about the *set* of digits used, not the count of positions.

Two more checks on the sample. Is 102 really the smallest three-digit number with three distinct digits? The candidates below it are 100 (digits $\{0,1\}$: only two), 101 (digits $\{0,1\}$: two). So yes, 102 is the first. Is 987 really the largest? The only three-digit numbers above it are 988 to 999. The numbers 988 and 989 use only the digits $\{8,9\}$, and every number from 990 to 999 begins with two 9s, so it has at most two distinct digits. None of them qualifies, so yes, 987 is the largest.

Why is $N \leq 18$? Because an 18-digit number fits in a 64-bit integer — but notice that we never need arithmetic on these numbers at all. We print them digit by digit as characters, so the size limit is irrelevant to our method. That is a small piece of good fortune worth noticing in Phase 1.

1.9.2 Phase 2 — Abstract

An N-digit number is compared with another N-digit number the way words are compared in a dictionary: look at the leftmost position where they differ, and whichever has the smaller digit there is the smaller number. This is because the leftmost digit carries weight 10^{N-1}, which is larger than everything to its right put together — the largest possible number formed by the remaining $N-1$ digits is $\underbrace{99\cdots9}_{N-1} = 10^{N-1}-1$, which is still less than one unit of the leading place.

> **WHY GREEDY IS SAFE HERE.** Because position 1 dominates positions 2 through N combined, choosing the smallest legal digit for position 1 can never be a mistake — no cleverness later can compensate. The same argument then applies to position 2 among positions $2..N$, and so on down the number. This is the greedy-choice property, and it is what licenses everything below.

Step 1 — the special case $K = 1$. With only one distinct digit allowed, the number is that digit repeated N times. The digit cannot be 0, because then the first digit would be a leading zero. So the smallest is $\underbrace{11\cdots1}_{N}$ and the largest is $\underbrace{99\cdots9}_{N}$. For $N = 1$ that gives 1 and 9, matching the sample.

Step 2 — the minimum when $K \geq 2$. Work left to right, always choosing the smallest digit we can still afford.

Position 1. It must not be 0. The smallest non-zero digit is 1. Take **1**. Digits used so far: $\{1\}$, that is 1 of our K.

Positions 2 onward. The smallest digit of all is 0, and 0 is new, so it is free to use. After using it we have $\{0,1\}$, two distinct digits, and we still owe $K-2$ more. Each remaining debt needs its own position. If the total length is N and position 1 is spent, we have $N-1$ positions left, of which the last $K-2$ must carry the still-unused digits. That leaves

$$(N-1)-(K-2) = N-K+1$$

positions that may be filled with 0. Fill all of them with **0** — taking fewer would force a larger digit into a more significant place, which is worse.

The last $K-2$ positions. They must carry $K-2$ digits, each different from 0 and 1 and from each other. To keep the number small we use the smallest available values, $2,3,\ldots,K-1$, and we place them in **increasing** order so that the smaller ones sit further left.

Putting it together:

$$\text{minimum} \;=\; 1\;\underbrace{00\cdots0}_{N-K+1}\;23\cdots(K-1). \tag{1.6}$$

Count the digits: $1+(N-K+1)+(K-2) = N$. Count the distinct digits: $\{1\}\cup\{0\}\cup\{2,\ldots,K-1\} = \{0,1,\ldots,K-1\}$, which has exactly K members.

Step 3 — the maximum when $K \geq 2$. The mirror image. Work left to right, always choosing the largest digit we can still afford.

Leading positions. The largest digit is 9, and there is no leading-zero worry. We may use 9 in as many positions as possible, but we must keep $K-1$ positions at the end for the other required digits. So the first $N-K+1$ positions all get **9**.

The last $K-1$ positions. They must carry $K-1$ digits, all different from 9 and from each other, and as large as possible. Those are 8,7,6, … down to $10-K$, placed in **decreasing** order so the larger ones sit further left.

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

Digit count: $(N-K+1)+(K-1)=N$. Distinct digits: $\{9\}\cup\{8,7,\ldots,10-K\}=\{10-K,\ldots,9\}$, exactly K of them.

HAND-WORKED VERIFICATION — $N=5$, $K=3$.

Minimum. Position 1 gets 1. Zeros to place: $N-K+1=5-3+1=3$, so three 0s. Remaining digits: $K-2=1$, namely 2. Concatenating: 1 then 000 then 2 gives **10002**. Digits used: {1,0,2} — three distinct, as required. Length five. Correct.

Maximum. Nines to place: $N-K+1=3$, so 999. Remaining digits: $K-1=2$, namely 8 then 7. Concatenating: 999 then 87 gives **99987**. Digits used: {9,8,7} — three distinct. Length five. Correct.

Figure 1.11 tabulates the two constructions, Equation (1.6) and Equation (1.7), for every N from 1 to 10 and every K from 1 to 5.

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 1.11 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.

Trace the row $K=5$ in **Figure 1.11** at $N=10$. The minimum reads 1 0 0 0 0 0 0 2 3 4. Our formula: one leading 1, then $N-K+1=10-5+1=6$ zeros, then $K-2=3$ digits 2,3,4. That is 1, six zeros, 234 — the very same ten digits. The maximum reads 9 9 9 9 9 9 8 7 6 5: six nines, then $K-1=4$ digits 8,7,6,5. Also identical. The table and the formula agree everywhere, which is the point of drawing the table.

KEY INSIGHT. Greedy digit placement: put the extreme digits (1 then 0 for the minimum, 9 for the maximum) in the most significant positions, and postpone the remaining required distinct digits to the least significant end.

1.9.3 Phases 3 and 4 — Design and Analyze

1. Read T. Repeat T times:
2. Read N and K.
3. If $K = 1$: print N copies of `1`, a space, N copies of `9`, a newline; move on.
4. Otherwise print `1`, then $N - K + 1$ copies of `0`, then the digits $2,3,\ldots,K-1$; print a space; then print $N - K + 1$ copies of `9`, then the digits $8,7,\ldots,10-K$; print a newline.

Correctness. Proved above by the greedy-choice argument, position by position, together with the digit counts that confirm the constructions have exactly N digits and exactly K distinct ones.

Time. Each test case prints $2N$ characters and does constant work per character: $O(N)$ per case, $O(TN)$ overall. With $T \approx 200$ and $N \leq 18$ that is about 7,200 character writes — instant.

Space. $O(1)$. Nothing is stored; the digits are printed as they are computed.

1.9.4 Phase 5 — Implement

Listing 1.11 The complete solution to *Minimum and maximum numbers*, SPOJ NEWLANG.

```
#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;
}
```

Two loops deserve a second look. `for (int i = 1; i <= k - 2; i++) printf("%d", i + 1);` prints $2,3,\ldots,K-1$ in increasing order, because as i runs $1,2,\ldots,K-2$ the printed value $i+1$ runs $2,3,\ldots,K-1$. And `for (int i = 1; i <= k - 1; i++) printf("%d", 9 - i);` prints $8,7,\ldots,10-K$ in decreasing order, because as i runs $1,\ldots,K-1$ the printed value $9-i$ runs $8,7,\ldots,10-K$. When $K = 2$ the first of these loops runs zero times, which is exactly right: no extra digits are needed beyond 1 and 0.

On the judge's sample input

```
2
1 1
3 3
```

Listing 1.11 actually printed

```
1 9
102 987
```

which matches **Figure 1.10** exactly. Three further cases were run as a check. For `5 3` it printed `10002 99987`, agreeing with the hand-worked verification above. For `10 5` it printed `1000000234 9999998765`, agreeing with the $K = 5$ row of **Figure 1.11**. For `10 2` it printed `1000000000 9999999998`, agreeing with the $K = 2$ row.

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

Paradigm: bit manipulation / mathematical pattern.

1.10.1 Phase 1 — Understand

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 → 01 → 0112 → 01121220 → …`

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

Figure 1.12 defines the sequence. It begins with the single digit 0. Then one action is repeated forever: **append the sequence you already have to its own right, but with every** 0 **turned into** 1**, every** 1 **into** 2**, and every** 2 **into** 0**.**

Written out, the first few stages are

$$0 \;\to\; 01 \;\to\; 0112 \;\to\; 01121220 \;\to\; \cdots$$

Let us make sure we can reproduce that by hand, because this is exactly the "small example worked with a pencil" of Phase 1.

Stage 0. The sequence is `0`. *Stage 1.* Take `0`. Shift it: 0 → 1, giving `1`. Append: `0` + `1` = `01`. *Stage 2.* Take `01`. Shift it: 0 → 1 and 1 → 2, giving `12`. Append: `01` + `12` = `0112`. *Stage 3.* Take `0112`. Shift it: 0 → 1, 1 → 2, 1 → 2, 2 → 0, giving `1220`. Append: `0112` + `1220` = `01121220`. *Stage 4.* Take `01121220`. Shift it, giving `12202001`. Append: `0112122012202001`.

The statement quotes the sequence as beginning `011212201220200112...`, and our stage-4 string `0112122012202001` followed by the first two characters of stage 5 gives `011212201220200112`. It matches.

The task. Given m, report the m-th digit of the sequence, counting from 1.

The constraints. The first line holds n, the number of queries, with $1 \le n \le 1000$. Each of the next n lines holds one m with $1 \le m \le 2^{63} - 1$.

That upper limit, $2^{63} - 1$, is about 9.22×10^{18}. Turn to **Table 1.3** again: nothing that even *counts up to m* can survive. We are being told there is a formula.

Interesting sequence (E-Olymp 4894)

Time limit 1 s Memory limit 128 MB

Sample

Input #1	Answer #1
3 17 239 385	1 0 2

Figure 1.13 The sample test of *Interesting sequence*: three queries 17, 239, 385, with answers 1, 0, 2.

1.10.2 Phase 2 — Abstract

Step 1 — name the transformation. The replacement $0 \to 1$, $1 \to 2$, $2 \to 0$ looks like a permutation of three symbols, but look at the numbers: $0 + 1 = 1$, $1 + 1 = 2$, $2 + 1 = 3$, and 3 wraps round to 0. The transformation is simply **add one, modulo three**. Writing $v \mapsto (v + 1) \bmod 3$ replaces a three-line table with one piece of arithmetic, and arithmetic is something we can reason about.

So: the second half of the sequence at each stage is the first half with every value increased by 1 modulo 3.

Step 2 — recursion on the binary representation. The sequence doubles in length each stage: after k stages it has 2^k digits. Index the digits from 0 rather than 1 for a moment; call the index p.

Suppose p lies in a sequence of length 2^{k+1}. There are two cases.

- If $p < 2^k$, the digit is in the **first half**, which is just the sequence of length 2^k. So the value is whatever it was one stage earlier.
- If $p \ge 2^k$, the digit is in the **second half**. Write $p = 2^k + q$ with $0 \le q < 2^k$. The second half is the first half shifted, so the value is (value at q) + 1 mod 3.

Now think about what those two cases look like in **binary**. The binary representation of a number writes it as a sum of powers of two; for example $13 = 8 + 4 + 1$, so in binary 13 is 1101, meaning one eight, one four, no two, one unit. For a number $p < 2^{k+1}$, asking "is $p \ge 2^k$?" is exactly asking "is the bit of weight 2^k — the leading bit — equal to 1?" And forming $q = p - 2^k$ is exactly "delete that bit".

So the recursion reads: *strip the bits of p one at a time from the top; each bit that is* 1 *adds one, modulo three; each bit that is* 0 *adds nothing.*

Step 3 — the closed formula. If each 1-bit contributes $+1$ and each 0-bit contributes nothing, and we start from the value 0 at $p = 0$, then

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

The number of 1-bits of a number is called its **population count**, or *popcount*.

A proof by induction of Equation (1.8), since Step 2 was reasoning and this is the claim. Let S_k denote the sequence after k stages, of length 2^k, and let $P(p)$ denote the popcount of p.

Base case. S_0 is the single digit 0 at index $p = 0$. The binary form of 0 has no 1-bits, so $P(0) = 0$, and $0 \bmod 3 = 0$. The claim holds.

Inductive step. Assume $S_k[p] = P(p) \bmod 3$ for all $0 \le p < 2^k$. Take any p with $0 \le p < 2^{k+1}$. If $p < 2^k$ then $S_{k+1}[p] = S_k[p] = P(p) \bmod 3$ by the assumption, and we are done. If $p \ge 2^k$, write $p = 2^k + q$ with $0 \le q < 2^k$. Because $q < 2^k$, the bit of weight 2^k in q is 0, so adding 2^k simply turns that bit on and disturbs nothing else. Hence $P(p) = P(q) + 1$. By the construction of the sequence, $S_{k+1}[p] = (S_k[q] + 1) \bmod 3 = (P(q) \bmod 3 + 1) \bmod 3 = (P(q) + 1) \bmod 3 = P(p) \bmod 3$. The claim holds for $k + 1$. By induction it holds for every k, hence for every index. ■

Step 4 — convert the indexing. The problem numbers the digits from 1; our formula numbers them from 0. So the digit at position m is the value at index $p = m - 1$.

> **KEY INSIGHT.** answer $= \text{popcount}(m - 1) \bmod 3$.

Figure 1.14 tabulates the first twelve indices, their binary forms, their popcounts, and the popcount modulo three.

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

Read the last column of **Figure 1.14** straight down: 0,1,1,2,1,2,2,0,1,2,2,0. Now read the first twelve digits of the sequence we built by hand in Phase 1, `0112122012202001`: they are 0,1,1,2,1,2,2,0,1,2,2,0. Identical. The table *is* the sequence.

> **HAND-WORKED VERIFICATION — the three sample queries.**
>
> *Query* $m = 17$. Then $p = 16$. In binary $16 = 10000$. Number of 1-bits: 1. $1 \bmod 3 = \mathbf{1}$.
>
> *Query* $m = 239$. Then $p = 238$. Convert to binary by repeatedly taking out the largest power of two: $238 - 128 = 110$; $110 - 64 = 46$; $46 - 32 = 14$; $14 - 8 = 6$; $6 - 4 = 2$; $2 - 2 = 0$. So $238 = 128 + 64 + 32 + 8 + 4 + 2$, which in binary is 11101110. Number of 1-bits: 6. $6 \bmod 3 = \mathbf{0}$.

> *Query* $m = 385$. Then $p = 384$. $384 - 256 = 128$; $128 - 128 = 0$. So $384 = 256 + 128$, in binary 110000000. Number of 1-bits: 2. 2 mod 3 = **2**.
>
> The three answers are 1, 0, 2 — exactly what **Figure 1.13** expects.

Three spot checks are reassuring but not conclusive. **Listing 1.12** builds the real sequence out to 128 terms by literally following the construction, then compares every single term against the formula.

Listing 1.12 Building *Interesting sequence* by brute force and checking the popcount formula against every term.

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

int main() {
    vector<int> s(1, 0);                        // the sequence starts as [0]
    for (int step = 0; step < 7; step++) {      // 7 doublings gives 128 terms
        vector<int> t = s;
        for (size_t i = 0; i < t.size(); i++) t[i] = (t[i] + 1) % 3;
        s.insert(s.end(), t.begin(), t.end());
    }
    printf("first 32 terms : ");
    for (int i = 0; i < 32; i++) printf("%d", s[i]);
    printf("\n");
    int bad = 0;                                // compare the formula with the real sequence
    for (size_t i = 0; i < s.size(); i++)
        if (s[i] != __builtin_popcountll(i) % 3) bad++;
    printf("terms checked  : %d, mismatches : %d\n", (int)s.size(), bad);
    return 0;
}
```

Its actual output is:

```
first 32 terms : 01121220122020011220200120010112
terms checked  : 128, mismatches : 0
```

The first eighteen characters, `011212201220200112`, are precisely the prefix printed in the problem statement of **Figure 1.12**, and not one of the 128 terms disagrees with the formula.

1.10.3 Phases 3 and 4 — Design and Analyze

1. Read n, the number of queries.
2. For each query, read m as a 64-bit unsigned integer, compute the popcount of $m - 1$, print it modulo 3.

Correctness. Proved by induction in Step 3, plus the index shift of Step 4.

Time. $O(1)$ per query. The popcount of a 64-bit value is a single machine instruction on modern processors, which GCC emits for `__builtin_popcountll`. Over $n \leq 1000$ queries the total work is negligible.

Space. $O(1)$. We never store the sequence — which is just as well, since after 63 doublings it would have 2^{63} terms.

1.10.4 Phase 5 — Implement

Listing 1.13 The complete solution to *Interesting sequence*, E-Olymp 4894.

```
#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;
}
```

The type `unsigned long long` holds whole numbers from 0 up to $2^{64} - 1$, which comfortably covers the limit $2^{63} - 1$; an ordinary `int`, which stops at about 2.1×10^9, would overflow catastrophically. `__builtin_popcountll` is GCC's name for the popcount of a 64-bit value — the trailing `ll` stands for *long long*. In C++20 the portable spelling is `std::popcount` from `<bit>`.

On the judge's sample input

```
3
17
239
385
```

Listing 1.13 actually printed

```
1
0
2
```

matching **Figure 1.13**. Asked for the first eight terms ($m = 1$ through 8) it printed `0 1 1 2 1 2 2 0`, which is the stage-3 sequence `01121220` we constructed by hand.

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

Paradigm: dynamic programming / memoization.

1.11.1 Phase 1 — Understand

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

Figure 1.15 defines a function f on the whole numbers:

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

The symbol $\lfloor x \rfloor$ in Equation (1.9) is the **floor** of x: the largest whole number that is not greater than x. In plain terms it means "divide and throw away the fractional part". So $\lfloor 7/2 \rfloor = 3$, $\lfloor 7/3 \rfloor = 2$, $\lfloor 1/2 \rfloor = 0$. In C++, integer division with `/` on non-negative values does exactly this, so `n / 2` already is $\lfloor n/2 \rfloor$.

The input. One integer n with $1 \le n \le 10^{18}$. **The output.** The value $f(n)$. **The limit.** One second.

The sample is $n = 5$ with answer 5.

> **HAND-WORKED EXAMPLE — the sample.** $f(5) = f(\lfloor 5/2 \rfloor) + f(\lfloor 5/3 \rfloor) = f(2) + f(1)$. $f(2) = f(1) + f(0)$. $f(1) = f(0) + f(0) = 1 + 1 = 2$. So $f(2) = 2 + 1 = 3$. So $f(5) = 3 + 2 = 5$. This is the expected answer.

1.11.2 Phase 2 — Abstract

Step 1 — the naive recursion is hopeless. Translate the definition into code directly and every call with an argument greater than 0 makes two more calls. How many calls happen altogether? We can count them exactly with a small picture.

Draw every call as a node of a tree. A call with argument $u > 0$ has exactly two children, the calls for $\lfloor u/2 \rfloor$ and $\lfloor u/3 \rfloor$. A call with argument 0 has no children; it is a *leaf*, and it returns 1. Every value in the computation is built by adding up those 1s, so $f(n)$ is exactly the number of leaves. Finally, a tree in which every inner node has exactly two children always has one inner node fewer than it has leaves (each inner node joins two subtrees into one, and we start with as many subtrees as leaves and finish with one). So the total number of calls is

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

Check Equation (1.10) on the sample. Since $f(5) = 5$, it predicts $2 \cdot 5 - 1 = 9$ calls. The hand-worked example of Section 1.11.1 makes the calls $f(5), f(2), f(1), f(0), f(0), f(0), f(1), f(0), f(0)$ — nine of them.

For the largest input, $f(10^{18}) \approx 2.24 \times 10^{14}$ (Section 1.11.3 computes it exactly), so the naive recursion makes about 4.5×10^{14} calls. Against a budget of 10^8 operations per second that is about 4.5×10^6 seconds — some fifty-two days. Unacceptable. (A quicker estimate, "two calls each, about $\log_2 10^{18} \approx 60$ levels deep, so 2^{60} calls", overshoots, because the $\lfloor n/3 \rfloor$ branches reach 0 after only about 38 levels. But it reaches the same verdict.)

But notice *why* it is so bad. Look again at the sample trace: $f(1)$ was needed once inside $f(2)$ and once more directly inside $f(5)$. Nothing new is being computed the second time. This is exactly the overlapping-sub-problem symptom from Section 1.4.4.

Step 2 — count how many *different* sub-problems there actually are. Starting from n, every value we can ever be asked about is reached by a sequence of halvings and thirdings. Such a value has the form

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

(The floors can in fact be taken in any order along the way, because $\lfloor \lfloor n/p \rfloor / q \rfloor = \lfloor n/(pq) \rfloor$ for positive whole numbers p and q; so Equation (1.11) is exact, not an approximation.) How large can a get? Once 2^a exceeds n the value is 0 and we stop, so $a \leq \log_2 n$. For $n = 10^{18}$ that is at most 60. Similarly $b \leq \log_3 n \approx 38$. So there are at most $61 \times 39 = 2{,}379$ pairs (a, b), and therefore at most about 2,400 distinct sub-problems — many of which coincide anyway, because different pairs (a, b) can give the same floor.

Two thousand four hundred, against 4.5×10^{14}. That is the whole insight.

Step 3 — memoize. Keep a table of already-computed values. Before computing $f(u)$, look in the table; if it is there, return it. Otherwise compute it, store it, and return it.

Which table? Not an array. An array indexed by n would need 10^{18} entries — at eight bytes each, eight million terabytes. We need a container that stores only the keys we actually meet, scattered anywhere in a huge range. `std::map<long long, long long>` does exactly this: it holds the entries we insert and nothing else, and looks up a key in $O(\log(\text{entries}))$ time.

> **KEY INSIGHT.** Although n may be as large as 10^{18}, only about a thousand distinct sub-problems ever arise. A hash map or an ordered map turns an impossible computation into an instant one. The algorithm did not change; the data structure did.

Listing 1.14 settles the count empirically rather than by estimate: it walks the recursion from the largest legal input and records every distinct value it ever reaches.

Listing 1.14 Counting the distinct sub-problems of f by exploring the recursion with a `set`.

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

set<ll> seen;

void walk(ll u) {
    if (seen.count(u)) return;          // this sub-problem was already reached
    seen.insert(u);
    if (u == 0) return;
    walk(u / 2);
    walk(u / 3);
}

int main() {
    ll n = 1000000000000000000LL;       // 10^18, the largest allowed input
    walk(n);
    printf("distinct sub-problems for n = %lld : %d\n", n, (int)seen.size());
    seen.clear(); walk(12);
    printf("distinct sub-problems for n = 12 : %d\n", (int)seen.size());
    return 0;
}
```

Its actual output is:

```
distinct sub-problems for n = 1000000000000000000 : 1012
distinct sub-problems for n = 12 : 7
```

One thousand and twelve. The estimate of "about 2,400" was an upper bound and the truth is smaller still, as upper bounds often are. Either number is trivially small.

HAND-WORKED WALKTHROUGH — $f(12)$, with the cache written down at each step.

$f(12) = f(6) + f(4)$ — neither is known yet, so descend into $f(6)$ first. $f(6) = f(3) + f(2)$ — descend into $f(3)$. $f(3) = f(1) + f(1)$ — descend into $f(1)$. $f(1) = f(0) + f(0) = 1 + 1 = 2$. **Cache** $f(1) = 2$. $f(3) = 2 + 2 = 4$. The second $f(1)$ came straight from the cache. **Cache** $f(3) = 4$. $f(2) = f(1) + f(0) = 2 + 1 = 3$, both from the cache. **Cache** $f(2) = 3$. $f(6) = 4 + 3 = 7$. **Cache** $f(6) = 7$. $f(4) = f(2) + f(1) = 3 + 2 = 5$ — **both from the cache, no recursion at all.** $f(12) = 7 + 5 = 12$.

Count the distinct values the walkthrough touched: 12,6,4,3,2,1,0 — seven of them, exactly as **Listing 1.14** reports. And the answer $f(12) = 12$ is a coincidence, not a pattern; $f(11) = 9$ and $f(13) = 12$.

Table 1.6 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 1.6 is the same walkthrough laid out as the table the program actually holds in memory. Read the last column to see the order in which the recursion filled it: the deepest values first, the original question last.

1.11.3 Phases 3 and 4 — Design and Analyze

1. Create an empty map and put the single entry $f(0) = 1$ into it.
2. Define `rec(u)`: if u is a key of the map, return the stored value; otherwise compute `rec(u/2) + rec(u/3)`, store it under u, and return it.
3. Read n, print `rec(n)`.

Correctness. By strong induction on u. For $u = 0$ the seeded value is the definition. For $u > 0$, both $\lfloor u/2 \rfloor$ and $\lfloor u/3 \rfloor$ are strictly less than u (since $u \geq 1$ implies $\lfloor u/2 \rfloor \leq u/2 < u$), so by the induction hypothesis the two recursive calls return the correct values of f at those arguments; their sum is $f(u)$ by definition. The cache only ever returns a value that was computed by this same rule, so caching cannot introduce an error.

Termination. Every recursive call has a strictly smaller non-negative argument, so the recursion cannot go on forever; it must reach 0.

Time. Let D be the number of distinct sub-problems, measured above as 1,012 for the worst case. Each is computed once, and each computation performs a constant number of map operations at $O(\log D)$ each. The total is $O(D \log D)$, which for $D \approx 10^3$ is a few tens of thousands of elementary steps. Expressed in terms of n, D is $O(\log^2 n)$, so the running time is $O(\log^2 n \cdot \log\log n)$.

Space. $O(D) = O(\log^2 n)$ map entries — about a thousand, or a few tens of kilobytes.

Does the answer fit? f grows, so we should check. Running the program on the largest legal input $n = 10^{18}$ gives $f(n) = 224{,}412{,}958{,}156{,}019$, about 2.2×10^{14}. A `long long` holds values up to about 9.2×10^{18}, so there is no overflow. This check is part of Phase 4, and skipping it is a common way to turn a correct algorithm into a *Wrong Answer*.

1.11.4 Phase 5 — Implement

Listing 1.15 The complete solution to *Recursive function 1*, E-Olymp 10296.

```
#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;
}
```

Notice how the base case is handled in **Listing 1.15**. There is no `if (u == 0) return 1;` inside `rec`. Instead `mp[0] = 1` is written into the cache before the first call, so the ordinary "is it cached?" test catches it. This is a small, characteristic piece of competitive-programming economy: one mechanism doing two jobs.

Run on the sample input `5`, **Listing 1.15** actually printed:

```
5
```

Run on `12` it actually printed `12`, agreeing with the hand-worked walkthrough. Run on `1000000000000000000` — the largest input the problem allows — it actually printed `224412958156019`, and returned immediately.

1.12 All Six Problems at a Glance

Table 1.7 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(column count, 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 $n = 10^{18}$

Look down the "Time" column of **Table 1.7**. Not one of the six solutions is worse than $O(n)$, and two of them do not depend on the input size at all. Not one of the six programs is longer than twenty lines. Yet none of the six is *easy* — each one required a genuine insight in Phase 2. That is

the shape of the subject: the difficulty lives in the thinking, and the thinking, when it succeeds, makes the code short.

1.12.1 Recognizing which paradigm applies

Pattern recognition is a skill, and like every skill it is built by seeing many examples. **Table 1.8** is a starter kit: a list of phrases that commonly appear in problem statements, and what each should make you suspect.

Table 1.8 Symptoms and diagnoses. When a problem statement contains a phrase on the left, the technique on the right is worth trying first.

If you see…	Think…	Example here
"Find the minimum / maximum of something"	One pass, running aggregate	*Columns*, *Journey*
"Modular arithmetic, divisibility, remainders"	GCD and number-theoretic properties	*Unique Numbers*
"Construct the optimal X"	Greedy, most significant position first	*Min/Max Numbers*
"The sequence doubles at each step"	Binary representation, popcount	*Interesting Sequence*
"$f(n)$ depends on $f(n/a)$ and $f(n/b)$"	Memoization with a map (sparse DP)	*Recursive Function 1*
"n is enormous but few distinct states arise"	Map-based DP, never an array	*Recursive Function 1*
"Visit everything and return" on a line	Geometry, not traveling salesman	*Journey*
"Color / label so that neighbors differ"	Lower bound plus explicit construction	*Columns*

These are heuristics, not laws. **Table 1.8** will not solve a problem for you; it will tell you where to point your attention first, which is most of what experience consists of.

1.13 Where These Problems Come From: the Online Judges

An **online judge** is a website that stores problems together with secret test data. You submit a program; the judge compiles it, runs it against every test, compares the output with the expected answer character by character, and returns a verdict. The important verdicts are *Accepted*, *Wrong Answer*, *Time Limit Exceeded*, *Memory Limit Exceeded*, *Runtime Error* and *Compilation Error*.

This is an unusually honest form of evaluation. The judge does not care how elegant your idea is. It does not give partial credit for effort. Either your program produces the right answers within the time limit on tests you have never seen, or it does not — and the tests you have never seen are exactly the point, because they are what forces you to reason about *all* inputs rather than the three in the statement.

This book draws its problems mainly from two judges, and **Table 1.9** introduces them together with two others that are worth knowing as you go further.

Table 1.9 The four online judges met in this book. Their sizes change every day; the figures are those reported on the dates shown.

Judge	Address	Size, as reported	What it gives this book
E-Olymp	`eolymp.com`	9,178 problems in the archive (September 2026) [3]	*Columns* (10281), *Journey* (10280), *Interesting sequence* (4894), *Recursive function 1* (10296)
SPOJ, the Sphere Online Judge	`spoj.com`	about 1.29 million registered users and 35.6 million submissions (front page, early 2026) [4]	*Unique Numbers* (DCEPC901), *Minimum and maximum numbers* (NEWLANG), and a public ranking of the best solutions to every problem
UVa Online Judge	`onlinejudge.org`	more than 4,300 problems (September 2026) [5]	The archive that *Competitive Programming 4* [6] indexes problem by problem
beecrowd (formerly URI Online Judge)	`beecrowd.com`	more than one million developers in over 100 countries, and over 2,500 problems (early 2026) [7]	A gentle beginners' section in English and Portuguese

E-Olymp [3] is a Ukrainian judge with a large, well-graded problem archive and an interface available in English. Four of this chapter's six problems live there: *Columns* is problem 10281, *Journey* is 10280, *Interesting sequence* is 4894 and *Recursive function 1* is 10296.

SPOJ [4], the Sphere Online Judge, is one of the oldest and largest judges; early in 2026 its front page reported about 1.29 million registered users and more than 35 million submissions. The other two problems of this chapter — *Unique Numbers* (DCEPC901) and *Minimum and maximum numbers* (NEWLANG) — are SPOJ problems. SPOJ also publishes, for each problem, a ranking of the fastest accepted solutions, which we shall make use of in Section 1.14.

The **UVa Online Judge** [8], with more than 4,300 problems [5], is the archive that *Competitive Programming 4* [6] indexes problem by problem; if you work through that book, this is where you submit. **beecrowd** [7], formerly the URI Online Judge, is the largest judge in Latin America, with a substantial beginners' section in English and Portuguese, and it is a gentle place to start if the E-Olymp and SPOJ problem sets feel steep.

> **HOW TO USE A JUDGE WELL.** Do not treat the judge as a compiler with a slow turnaround. Every submission should follow all five phases. Work the sample by hand *before* you submit; if your hand answer differs from the sample, do not submit at all. When you get *Wrong Answer*, do not shuffle the code and resubmit — go back to Phase 2 and ask which input class you have not thought about. When you get *Time Limit Exceeded*, do not micro-optimize the loops — go back to Phase 4 and recompute the complexity against **Table 1.3**. The judge is a test of your reasoning, and treating it as a test of your typing wastes it.

1.14 Judge Problems as Research: Two Papers from ITS

It would be easy to dismiss online-judge problems as puzzles. They are not. The Department of Informatics at ITS has published peer-reviewed research whose experimental platform is precisely an online judge, and two examples are worth looking at now, in your first week, because they show where this subject leads.

1.14.1 Extrapolating a periodic function

PERIOD5 — Periodic function, trip 5 (SPOJ)

Category classical Time limit 1–7.5 s Source limit 50 000 B Memory limit 1536 MB

Task (summary)

A *clean periodic function* is a sum of periodic components whose periods are whole numbers. You are given the first values of such a function and must extrapolate it: predict the values that follow.

Motivation quoted on the problem page

"Solar cycle predictions are used by various agencies and many industry groups. The solar cycle is important for determining the lifetime of satellites in low-Earth orbit, as the drag on the satellites correlates with the solar cycle [...]" (NOAA). The page illustrates this with centuries of sunspot counts.

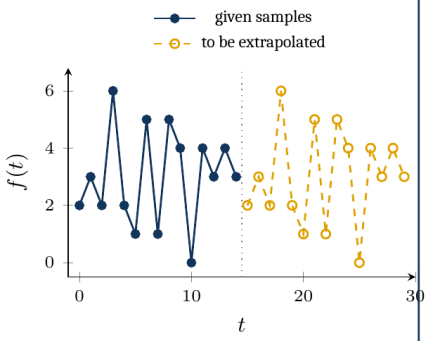


Figure 1.16 SPOJ problem PERIOD5, *Periodic function, trip 5*: given samples of a clean periodic function, extrapolate it. The small plot shows such a function, built from components of periods 3 and 5.

Figure 1.16 shows SPOJ problem PERIOD5. A function is built as a sum of periodic components with whole-number periods; you are given part of it and must predict the rest. The application quoted on the problem page is the solar cycle, which matters because satellite drag in low Earth orbit tracks sunspot activity.

Soelaiman et al. [9] attacked this with a system of linear equations, comparing an unoptimized formulation on an $N^2 \times N^2$ matrix with an optimized one on an $N \times N$ matrix. **Figure 1.17** summarizes the published paper.

Journal of Theoretical and Applied Information Technology, vol. 97, no. 8, pp. 2225–2239, 30 April 2019

Efficient Approach for Deterministic Data Extrapolation from a Clean Periodic Function with Periodic Components Representation by a System of Linear Equations

R. Soelaiman, S. Candra, M. M. I. Subakti
Department of Informatics, Institut Teknologi Sepuluh Nopember (ITS), Surabaya, Indonesia

In brief

The paper compares an unoptimized and an optimized way to extrapolate a clean periodic function whose components have integer periods. Both identify the components by solving a system of linear equations; the optimized formulation works on an $N \times N$ system instead of an $N^2 \times N^2$ one, so it runs faster, allocates less memory and handles a larger range of periods. Correctness was tested by submission to the Sphere Online Judge, and performance by measuring running times.

Figure 1.17 Paper [9] at a glance: title, byline, venue (*Journal of Theoretical and Applied Information Technology*, vol. 97, no. 8, 2019) and what it contributes.

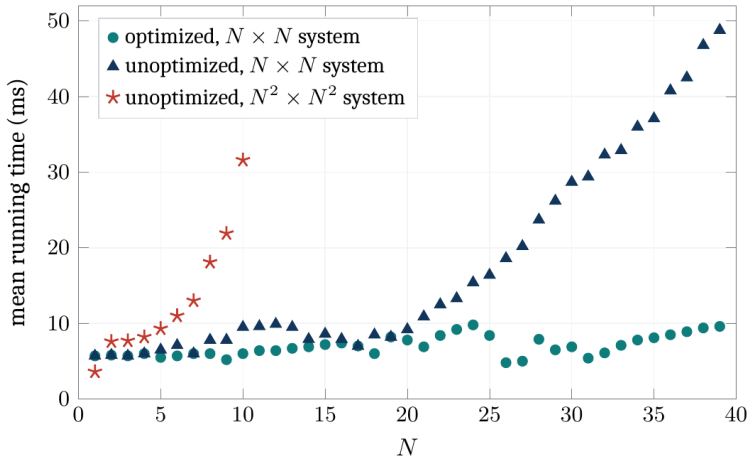


Figure 1.18 Mean running time of three approaches as N grows, redrawn from the measurements in Figure 13 of [9]. The optimized $N \times N$ method stays flat; the others climb.

Read **Figure 1.18** with the eyes of Section 1.3.4. The star-marked series, the unoptimized $N^2 \times N^2$ method, leaves the top of the chart before $N = 11$. The triangle-marked series, the unoptimized $N \times N$ method, is fine until about $N = 20$ and then climbs steeply. The circle-marked series, the optimized method, is almost a horizontal line all the way to $N = 39$. Three correct algorithms; one of them usable.

PERIOD5: statistics and best solutions

Users accepted	Submissions	Accepted	Wrong answer	Compile error	Runtime error	Time limit exceeded
12	424	235	84	9	57	39

Rank	Date	User	Result	Time (s)	Memory	Language
1	2018-09-01	Steven Candra	accepted	0.59	17M	CPP14
2	2020-01-26	Rully Soelaiman	accepted	0.59	4.9M	CPP
3	2017-05-17	Mateusz Radecki	accepted	0.60	94M	CPP14
4	2017-05-13	Kamil Debowski	accepted	0.72	17M	CPP14
5	2017-05-23	i1e2s	accepted	0.73	17M	CPP14
6	2017-05-24	Balsa	accepted	0.76	88M	CPP14
7	2017-09-23	[Rampage] Blue.Mary	accepted	0.79	3.8M	C++ 4.3.2
8	2017-09-11	[Rampage] Blue.Mary	accepted	0.81	17M	CPP
9	2017-05-04	Domagoj Bradac	accepted	0.82	20M	CPP14

As listed on the SPOJ ranking page in early 2026; the highlighted row is the ITS entry.

Figure 1.19 The SPOJ ranking for PERIOD5 as listed in early 2026. Of four hundred and twenty-four submissions only two hundred and thirty-five were accepted, and the ITS entry sits at rank 2 with a time of 0.59 seconds in 4.9 MB.

Figure 1.19 is how the paper measured itself: not with a claim, but with a public ranking on a judge that anybody can check. Note the memory column — the ITS solution uses 4.9 MB where the entry above it uses 17 MB. That is the optimized matrix formulation showing up as a measurable number.

1.14.2 Reading handwritten digits

HIR — Hard Image Recognition (SPOJ)

Category challenge Time limit 0.1 s Source limit 150 000 B Memory limit 1536 MB

Statement

Write a program that recognizes images of six-digit numbers. In some cases the image may be blurred or deformed, but a human can always read the number written. The picture is black and white: a white background with a black raster image on it, where the character '`.`' denotes white and 'X' black. (The page tells the story of Byteland, whose passport numbers are written in every font a word processor offers.)

Input

t, the number of test cases ($t \leq 250$), then t tests. Each test starts with two integers H and W, the height and width of the picture ($10 \leq H, W \leq 200$), followed by H rows of W characters each ('`.`' or 'X').

Output

For each test case, output the recognized six-digit number on a separate line. This is a *challenge* problem: the score is the number of images read correctly.

Figure 1.20 SPOJ problem HIR, *Hard Image Recognition*: recover a six-digit number from a black-and-white raster image, given a 0.1-second limit and a 150 kB source-code limit.

Figure 1.20 shows a very different problem. You receive up to two hundred and fifty pictures, each up to 200 characters wide and drawn with `.` for white and `X` for black, and you must read the six-digit number written in each. **Figure 1.21** shows how varied the fonts are.

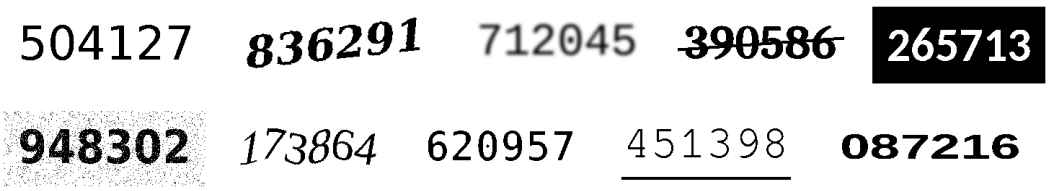


Figure 1.21 Ten six-digit numbers drawn in different fonts, some tilted, blurred, struck through, speckled or inverted — the kind of variety the HIR sample images show.

HIR — sample test

Input

```
2
13 40
.........................................
.........................................
.........................................
...XXX...XXX...XXX.....XX..XXXX..XXXX...
..XX.XX.XX.XX.XX.XX...XXX..XX.XX.X..XX..
..X..XX.XX..X.XX..XX.XXXX..XXXX.....XX..
..X..XX.XX..X.XX..XX.X.XX..XXXX....XX...
..X..XX.XX.XX.XX..X.XXXXXX.X..XX..XX....
..XXXXX.XX.XX.XXXXX....XX..XX.XX.XXXXX..
...XXX...XXX...XXX.....XX...XXX..XXXXX..
.........................................
.........................................
.........................................
12 60
............................................................
............................................................
.............XXXX...XXX..XXXXX.XXXX..XXXX..XXXX.............
.............X.....XX....X..X..X..X....XX....XX.............
.............X....XX........X..X..X....XX....XX.............
.............XXXX.XXXXX....XX..XXXX...XX....XX.............
................X.XX..XX...X...X..X....XX....XX.............
................XXXX..XX..XX..XX..XX....XX....XX.............
............X...X..X..XX..X...XX..XXX...XXX...XX.............
.............XXXX...XXX..XX....XXXX..XXXX..XXXX.............
............................................................
............................................................
```

Output

```
000482
567833
```

Score

A submission that prints `000482` and `567843` scores 1: the first answer is correct, the second has one incorrect digit.

Figure 1.22 The sample test of HIR. The two pictures encode 000482 and 567833; the submission shown scores 1 because its second answer has one incorrect digit.

Figure 1.22 shows the scoring rule, which is unusual: this is a *challenge* problem, so you are not simply accepted or rejected but given a score, one point per fully correct image. Soelaiman et al. [10] used a genetic algorithm to design the architecture of a neural network for this task — that is, an algorithm that searches for a good algorithm — subject to the brutal constraints of a 0.1-second time limit and a 150 kilobyte source limit, which rule out shipping a large pre-trained model. **Figure 1.23** summarizes the paper and **Figure 1.24** shows the result.

Journal of Theoretical and Applied Information Technology, vol. 98, no. 17, pp. 3479–3492, 15 September 2020

Genetic Algorithm Approach for Automated Generation of Neural Networks Architecture for Robust Digit Recognition

R. Soelaiman, Y. H. Pratama, M. M. I. Subakti, Y. Purwananto
Department of Informatics, Institut Teknologi Sepuluh Nopember (ITS), Surabaya, Indonesia

In brief

The paper designs a solution to the SPOJ problem *Hard Image Recognition* (HIR): reading six-digit numbers from noisy raster images under a very short time limit and a small source-code limit. Instead of choosing the architecture of the artificial neural network by trial and error, a genetic algorithm searches for it; together with a purpose-built training set, the result satisfies the constraints and reaches a score of 108 on SPOJ HIR.

Figure 1.23 Paper [10] at a glance: title, byline, venue (*Journal of Theoretical and Applied Information Technology*, vol. 98, no. 17, 2020) and what it contributes.

HIR: statistics and best solutions

Users accepted	Submissions	Accepted	Wrong answer	Compile error	Runtime error	Time limit exceeded
22	660	445	0	35	117	63

Rank	Date	User	Result	Time (s)	Memory	Language
1	2020-01-09	Yolanda Hertita	**108**	0.12	5.1M	CPP14
2	2020-09-16	Rully Soelaiman	**108**	0.15	5.0M	CPP
3	2018-12-25	CynthiaDewi	**32**	0.07	17M	CPP14
4	2025-05-22	dasd	**9**	0.05	5.2M	CPP14
5	2010-08-24	.::Manish Kumar::.	**2**	0.00	2.2M	C
6	2011-01-21	Satyarth Kumar Prasad	**2**	0.00	2.6M	C++ 4.3.2
7	2025-05-27	邓紫棋	**2**	0.04	5.2M	CPP14

As listed on the SPOJ ranking page in early 2026; the highlighted rows are the ITS entries.

Figure 1.24 The SPOJ ranking for HIR as listed in early 2026. Twenty-two users accepted out of six hundred and sixty submissions; the two top entries, scoring 108, are from the ITS group.

The lesson of **Figure 1.24** is not that neural networks are exciting. It is that the constraints in the problem statement — 0.1 seconds, 150 kB — dictated the entire design, exactly as **Table 1.3** says they should. Phase 1 decided the outcome of a research paper.

1.15 The Books, the Road Map, and the Evaluation Chapters

1.15.1 The books

Two families of books stand behind this one, and **Table 1.10** lists them with the editions that are current in September 2026.

Table 1.10 The two families of books behind this one, with the editions current in September 2026.

Book	Written by	Edition	Role in this book
Introduction to Algorithms (CLRS) [11]	Cormen, Leiserson, Rivest, Stein	4th ed., MIT Press, April 2022 — the current edition [12]	The primary reference: proofs and analysis
Introduction to Algorithms (CLRS)	Cormen, Leiserson, Rivest, Stein	3rd ed., MIT Press, 2009	Still common in university libraries; chapter numbers differ
Competitive Programming 4 (CP4) [6]	Halim, Halim, Effendy	Book 1 (Chapters 1–4) and Book 2 (Chapters 5–9), July 2020 — the current edition [13]	The practical companion: techniques and judge problems
Competitive Programming 3 (CP3)	Halim, Halim	3rd ed., 2013	The edition many students first meet

CLRS — Cormen, Leiserson, Rivest and Stein — is the standard reference for algorithm design and analysis worldwide. This book follows the fourth edition [11], published in April 2022 and still the latest edition in September 2026 [12]; if you have access only to the third edition, the chapter numbering differs but almost all the material is the same. The fourth edition is the one to buy if you are buying.

CP4 — Halim, Halim and Effendy [6], published in two volumes in July 2020, with *Competitive Programming 3* (2013) before it — is the practical companion. The CP4 team releases a new edition only when the changes justify one, and no fifth edition had appeared by September 2026 [13]. Where CLRS proves, CP4 drills: it catalogs techniques and points you at judge problems that exercise each one. The two books are complementary, and this book draws on both. Read CLRS to understand why an algorithm works; read CP4 to get faster at recognizing which one to reach for.

1.15.2 The road map

Table 1.11 The sixteen chapters of this book, with the corresponding chapters of [11].

Ch	Topic	CLRS
1	Foundations, framework and first problems	Ch. 1
2	Getting started: the RAM model, insertion sort, searching	Ch. 2
3	Correctness, asymptotic analysis, Big-O, solving recurrences	Ch. 2–4
4	**Evaluation and Discussion I** (Chapters 1–3)	—
5	Sorting: heapsort and quicksort	Ch. 6–7
6	Binary search trees and AVL trees	Ch. 13
7	Dynamic programming	Ch. 14
8	**Evaluation and Discussion II** (Chapters 5–7)	—
9	Greedy algorithms	Ch. 15
10	Elementary graph algorithms	Ch. 20
11	Minimum spanning trees	Ch. 21
12	**Evaluation and Discussion III** (Chapters 9–11)	—
13	Single-source shortest paths	Ch. 22
14	Amortized analysis	Ch. 16
15	All-pairs shortest paths	Ch. 23
16	**Evaluation and Discussion IV** (Chapters 13–15 and the whole book)	—

Table 1.11 is worth studying for a minute, because it shows the shape of the whole book. The first three chapters build the toolkit — framework, model of computation, asymptotic analysis.

Every fourth chapter pauses to evaluate and discuss what came before it. Everything else is a particular family of algorithms, and each family is one of the five paradigms of **Table 1.4** specialized to a class of problems. Nothing in the later chapters is disconnected from what you have read today.

1.15.3 The evaluation chapters

Four chapters of the book bring no new topic. Instead, each one sets problems on the three chapters just finished and then works every problem in full, so that you can test yourself first and check your reasoning afterward.

Table 1.12 The four evaluation chapters and what each one covers.

Chapter	Title	What it covers
4	Evaluation and Discussion I	Chapters 1–3: framework, correctness, asymptotics, recurrences
8	Evaluation and Discussion II	Chapters 5–7: sorting, search trees, dynamic programming
12	Evaluation and Discussion III	Chapters 9–11: greedy algorithms, graphs, spanning trees
16	Evaluation and Discussion IV	Chapters 13–15, and the whole book

Every chapter in **Table 1.12** asks you to *design and analyze*, not to recall. A typical problem there hands you a situation you have never seen and asks for the five phases: what is the input and output, what is the mathematical core, what is the algorithm, why is it correct, what is its complexity. Each problem is then solved in full, every step shown.

1.16 Setting Up Your Workbench: VS Code and a C++ Compiler

Every program in this book is meant to be typed, compiled and run by you. Before the next chapter asks you to do that, let us build the workbench once, carefully, on whichever computer you own. You need exactly two things:

1. **A compiler** — the program that turns the C++ you write into a program your computer can run. We use **GCC** (the `g++` command) on Windows and Linux, and **Clang** (the `clang++` command) on macOS. Both understand standard C++17, which is all this book uses. Any recent release will do; in September 2026 the newest are GCC 16.2 (August 2026) [14] and Clang 22.1 (June 2026) [15]. Recent GCC releases compile as C++20 unless told otherwise [14], which is why every command in this book passes `-std=c++17` explicitly.
2. **An editor** — the place where you write. We use **Visual Studio Code** (VS Code), a free editor from Microsoft that runs on Windows, macOS and Linux and looks the same on all three.

The compiler does the real work; the editor makes it comfortable. Install the compiler first, then the editor, then connect the two. **Table 1.13** gathers the whole procedure for the three operating systems on one page, so that you can come back to it later.

Table 1.13 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`

1.16.1 Windows, step by step

Windows does not come with a C++ compiler, so we borrow one from **MSYS2**, a small, well-maintained collection of the same tools that Linux programmers use.

1. Open a web browser, go to **msys2.org**, download the installer and run it. Accept the suggested folder `C:\msys64`. When it finishes, it offers to open a terminal; let it.
2. From the Start menu open **MSYS2 UCRT64** (the name matters: UCRT64 is the modern flavor). Type the following line and press Enter: `pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain`. When it asks which members to install, press Enter to accept all, and answer `Y` to proceed. The download takes a few minutes.
3. Tell Windows where the compiler lives. Press the Windows key, type *environment variables*, choose **Edit environment variables for your account**, select **Path**, click **Edit → New**, and type `C:\msys64\ucrt64\bin`. Click **OK** on every window.
4. Open a *new* Command Prompt or PowerShell window (old windows do not see the new Path) and type `g++ --version`. If a version number appears, the compiler is installed. If Windows says it does not recognize `g++`, step 3 was not saved; do it again.

1.16.2 macOS, step by step

macOS already knows where to get a compiler; it only needs permission.

1. Open **Terminal** (press `Cmd+Space`, type *Terminal*, press Enter).
2. Type `xcode-select --install` and press Enter. A window appears and asks whether to install the *command line developer tools*; click **Install** and wait.

3. Type `clang++ --version`. A version line means you are ready. On a Mac the command `g++` also works, but it quietly runs Clang; everything in this book compiles with both.

1.16.3 Linux, step by step

Most Linux systems install the compiler with one command.

1. Open a terminal (`Ctrl+Alt+T` on Ubuntu).
2. Type `sudo apt update && sudo apt install build-essential gdb`, press Enter, type your password, and confirm with `Y`.
3. Type `g++ --version` to check.

1.16.4 VS Code on every system

1. Download VS Code from **code.visualstudio.com** and install it as **Table 1.13** describes. On Windows, tick **Add to PATH** during installation, so that you can later type `code .` in a terminal to open a folder.
2. Start VS Code, open the **Extensions** view with `Ctrl+Shift+X` (`Cmd+Shift+X` on a Mac), search for **C/C++ Extension Pack**, and install the one published by Microsoft. It adds syntax coloring, error squiggles, code completion and the debugger.
3. Make a folder for this book, for example `DAA` in your Documents folder, and open it with **File → Open Folder**. Keep one subfolder per chapter — `Chapter01`, `Chapter02`, and so on — exactly as the accompanying `Code` folder is organized.

1.16.5 Your first program, compiled and run

Create a new file called `hello.cpp` in your folder, type in **Listing 1.16**, and save it.

Listing 1.16 The first program: it proves that the compiler, the editor and the terminal all work together.

```
#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;
}
```

Now open the terminal inside VS Code with **Terminal → New Terminal** (or press `Ctrl` together with the backtick key) and type the compile command from row 6 of **Table 1.13**:

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

Each part of that line has a job. `-std=c++17` selects the C++17 language standard used throughout this book; `-O2` asks the compiler to optimize, so that timing experiments measure the algorithm rather than a lazy translation; `-Wall` switches on warnings, which catch many mistakes before they become bugs; `-o hello` names the resulting program. If the compiler prints nothing, it succeeded — silence is good news. Run the program (`.\hello.exe` on Windows, `./hello` elsewhere) and type a name:

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

The last line is a small proof: `201703` is the value the C++ standard gives the `__cplusplus` macro under C++17, so you now know for certain which language version your programs are compiled against.

1.16.6 Feeding a judge problem its input

Online judges give a program its input through *standard input*. When a problem's sample input is long, do not type it by hand every time. Save it in a file, say `input.txt`, next to the program, and **redirect** it:

```
./columns < input.txt            (macOS, Linux, Windows Command Prompt)
Get-Content input.txt | .\columns.exe      (Windows PowerShell)
```

The program reads the file exactly as if you had typed it, which is exactly what the judge will do. Comparing your output with the sample output is then a matter of looking at two short blocks of text.

1.16.7 Debugging: watching the program think

When a program gives the wrong answer, the fastest way to find out why is to *watch* it. Click to the left of a line number in VS Code: a red dot appears — a **breakpoint**. Press `F5` and choose **C++ (GDB/LLDB)** and your compiler; VS Code creates the small configuration files it needs (`tasks.json` and `launch.json` inside a `.vscode` folder) and starts the program. It stops at the red dot, and the **Variables** panel on the left shows the value of every variable at that moment. Press `F10` to execute one line, `F11` to step into a function, `F5` to continue. Watching the loop invariant of Chapter 2 hold, iteration after iteration, in the Variables panel is one of the best ways to understand it.

> **WHEN SOMETHING GOES WRONG.** *"g++ is not recognized"* (Windows): the Path entry of step 3 is missing or the terminal was opened before you added it — open a new terminal. *"fatal error: iostream: No such file"*: the compiler was installed from the wrong MSYS2 terminal; repeat step 2 in **MSYS2 UCRT64**. *The program window flashes and closes*: run it from the terminal, not by double-clicking it. *Error squiggles everywhere in VS Code but the program compiles*: open the Command Palette (`Ctrl+Shift+P`), run **C/C++: Select IntelliSense Configuration**, and pick the compiler you installed.

With the workbench standing, every listing in the rest of this book is one copy, one compile and one run away from being yours.

1.17 Profitina in Practice

Profitina is an **AI visibility (GEO) platform**, live at `app.profitina.com`. GEO — Generative Engine Optimization — is the discipline of being *quotable by an answer engine*, as SEO was the discipline of ranking in a list of links. The product scans a brand's website, scores how ready its content is to be cited by an AI answer engine, explains each signal it measured, and then helps fix what it found.

Every idea in this chapter turns up in that product within the first week of engineering work.

Constraints decide the design, before any code. A scan touches every page of a site. A modest brand site has a few hundred pages; a marketplace has hundreds of thousands. Call the number of

pages P. Checking whether one page's main content is extractable is linear in the page's length, so the extraction pass over a whole site is $O(\text{total bytes})$ — unavoidable and fine. But a naive check for *near-duplicate pages*, comparing every page with every other, is $\Theta(P^2)$. At $P = 50{,}000$ that is 2.5×10^9 comparisons, which **Table 1.3** says is out of budget by an order of magnitude before we even account for the cost of comparing two documents. The fix is exactly the move of Section 1.4.5: replace the search with arithmetic. Hash each page's normalized content into a fingerprint and group by fingerprint, and the pass becomes $O(P)$. Same answer, different asymptotic class.

A running aggregate is usually enough. The historical analytics dashboard shows a weekly score trend per brand. The naive implementation recomputes each week's aggregate from the whole history; the incremental implementation of Section 1.4.1 keeps a running total and updates it as each new scan lands. On a table that grows every week, the first is $\Theta(\text{weeks}^2)$ over the life of the account and the second is $\Theta(\text{weeks})$.

Scheduling is a priority-queue problem. Paid tiers re-scan at a faster cadence than free ones. "Which site do I scan next?" is precisely "give me the item with the smallest next-due time", which is the one operation a heap is built for. Chapter 5 develops heaps properly; the point here is that the choice of data structure, not the choice of algorithm, is what makes the scheduler cheap.

Correctness has a legal meaning too. The platform is subject to Indonesia's Personal Data Protection Law, UU PDP 27/2022 [16]. Consent must be explicit, data must be exportable, and an account purge must actually purge. "Correct for every valid input" in Section 1.3.1 includes the input where a user asks to be deleted. And Profitina's fourth rule of product design applies to any algorithm you write here: **the app advises, the user decides.** A score is a measurement, never a guarantee.

1.17.1 The worked case study: Profitina Laundry

The hands-on case study in this book is always **Profitina Laundry**, a small teaching business, never the real product. Here is its chapter-1 problem, and the pleasing thing is that it is two of our six problems wearing an apron.

Profitina Laundry has n drop-off points along one street, Jalan Kertajaya, each at a known meter mark. A van leaves the shop, must call at every drop-off point, and must return to where it started. That is *Journey*: the shortest possible route is $2 \times (\text{furthest} - \text{nearest})$, and the shop may sit anywhere on the street without changing the answer.

Back at the shop, clean laundry is stacked on a drying rack: one column per drop-off point, as many baskets in a column as that point sent in. Two baskets in the same column must carry different labels, so that a worker reaching into a stack cannot confuse them; and two baskets side by side on the same shelf must carry different labels, unless a gap separates them. That is *Columns*: the number of labels needed is $\max(n, \text{tallest stack})$.

Listing 1.17 solves both at once.

Listing 1.17 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;
}
```

Given the input

```
5
120 340 75 980 410
3 7 2 9 4
```

— five drop-off points at meter marks 120,340,75,980,410, sending back 3,7,2,9,4 baskets — **Listing 1.17** actually printed:

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

Check both by hand. The nearest point is at 75 m and the furthest at 980 m, so the range is 905 m and the round trip is $2 \times 905 = 1810$ m. The rack has 5 columns and the tallest stack is 9 baskets, so $\max(5,9) = 9$ labels. Two research-grade insights, applied to a laundry, in twenty lines.

1.18 Common Mistakes, and How to Avoid Them

Writing code before understanding the problem. The single most expensive mistake in the subject. Symptom: you have two hundred lines and no idea which of them is wrong. Cure: close the editor, take a sheet of paper, and do Phases 1 and 2.

Testing instead of proving. "It works on the sample" tells you almost nothing, as Section 1.3.1 argued. Cure: for every loop, state the invariant; for every greedy choice, prove it is safe; for every recursion, use induction.

Ignoring the constraints. The constraints are the specification of your time budget. Reading $n \leq 10^{18}$ and writing a loop over n is not an oversight, it is a refusal to do Phase 4. Cure: before designing, put the largest allowed input into **Table 1.3** and decide what complexity you are allowed.

Optimizing the wrong thing. Replacing `cin` with `scanf` will not rescue an $O(n^2)$ algorithm at $n = 10^6$. Constant factors move you by a factor of two or five; asymptotic class moves you by a factor of a million. Cure: when you get *Time Limit Exceeded*, go back to Phase 2, not to the loop body.

Forgetting the edge cases the constraints permit. In *Unique Numbers*, x may be 0. In *Journey*, n may be 1, and two cities may share a coordinate. In *Min/Max Numbers*, K may be 1, which breaks the general formula and needs its own branch. Cure: read the constraint line, and for each quantity ask explicitly what happens at its smallest and its largest legal value.

Integer overflow. An `int` in C++ holds values only up to about 2.1×10^9. *Interesting sequence* allows m up to $2^{63} - 1$ and *Recursive function 1* allows n up to 10^{18}; both need a 64-bit type. Cure: in Phase 4, compute the largest value any variable can take, and pick the type from that.

Choosing the wrong container. An array indexed by n is unbeatable when n is small and impossible when n is 10^{18}. Cure: ask how many *distinct states* arise, not how large the index can be. In Section 1.11 the answer was 1,012.

Confusing "sorted" with "correct". The [1,2,3,4,5] counter-example of Section 1.3.2 is not a joke; students really do write checkers that test only condition (i). Cure: write down both conditions before writing the algorithm.

Mis-reading a defined term. *Substring* in *Columns* means a gap-free run, and everything depends on it. Cure: in Phase 1, restate every term the problem defines, in your own words, and test your restatement against the sample.

Treating the judge as a compiler. Twenty submissions in an hour means twenty guesses. Cure: one submission should follow one complete pass through the five phases.

1.19 Chapter Summary

An **algorithm** is a finite, definite, effective procedure taking a defined input to a defined output; a **program** is that algorithm expressed in a language a machine can run; a **data structure** is how the data is organized so the steps can be cheap. The algorithm is the idea, and it is the part that lasts.

An algorithm must be both **correct** — right for every input the constraints allow, which can be established only by proof — and **efficient**. Correctness is proved by direct argument, by contradiction, by **loop invariant** (true at entry, preserved by each pass, useful at exit), or by **induction**. Insertion sort was proved correct by its invariant in Section 1.3.3: at the start of pass j, the prefix $a[0 \ldots j-1]$ holds the original occupants of those positions, in order.

Efficiency is measured by growth rate. The operating rule is that a computer performs about 10^8 to 10^9 simple operations per second, so **the constraints in the problem statement decide the design before any code is written**. An input limit of 10^{18} does not mean "write a fast loop"; it means "there is no loop".

There are five design paradigms: **incremental / brute force**, **greedy**, **divide and conquer**, **dynamic programming**, and **mathematical / number-theoretic**. Recognizing which one a problem wants is the central skill, and **Table 1.8** is a starting catalog of symptoms.

Every problem is solved in five phases: **Understand, Abstract, Design, Analyze, Implement** — in that order, with the code last. The six problems of this chapter each took a page of thought and under twenty lines of C++: *Columns* is $\max(n, \max a_i)$; *Journey* is $2(x_{\max} - x_{\min})$; *Unique Numbers* divides a running n by $\gcd(n, x)$ on each multiplication; *Minimum and maximum numbers* places extreme digits greedily from the most significant end; *Interesting sequence* is $\text{popcount}(m - 1) \bmod 3$; and *Recursive function 1* memoizes about a thousand sub-problems in a `map` despite n reaching 10^{18}.

The abstraction phase is where the work happens. Its characteristic move is to prove a **lower bound** and then **construct** something that attains it; when both halves succeed, the answer is pinned exactly, and there is nothing left to test.

1.20 Exercises

1.1 Run the *Columns* solution on $n = 1$ with the single height 1000. What does it print, and which of the two constraints is doing the work?

1.2 *Columns* with seven columns, each of height 1. Predict the answer before running anything, then explain in one sentence.

1.3 In the proof of *Columns* we colored the cube at column j, row r with $((j + r) \bmod C) + 1$. Explain, without using the word "obviously", why two cubes in the same gap-free horizontal run can never receive the same color.

1.4 *Journey* with three cities all at coordinate 5. What is the answer, and does the formula still apply?

1.5 Modify *Journey* so that Cossack Vus need **not** return to his starting city. Derive the new answer, and say which starting city he should choose.

1.6 Work *Unique Numbers* by hand for $n = 36$ and the four operations `1 10`, `0 7`, `1 6`, `1 4`. Show each gcd computation.

1.7 Prove carefully that the operation "replace each i by $(i + x) \bmod n$" never changes the number of distinct values, for any $x \geq 0$.

1.8 The constraints of *Unique Numbers* allow $x = 0$ in a multiplication. What should the answer be, what does the formula give, and can the count ever rise again afterward?

1.9 Give the smallest and largest 4-digit numbers with exactly 4 distinct digits, by hand, then check against **Listing 1.11**.

1.10 Give the smallest and largest 18-digit numbers with exactly 1 distinct digit.

1.11 Find the 100th term of *Interesting sequence* by hand. Show the binary conversion.

1.12 Prove that no term of *Interesting sequence* is ever a digit other than 0, 1 or 2.

1.13 Compute $f(10)$ for *Recursive function 1* by hand, listing every cache entry in the order it is created.

1.14 Why can *Recursive function 1* not use an array for memoization? Give a number.

1.15 A problem gives $n \leq 10^5$ and a one-second limit. Which of $O(n)$, $O(n\log n)$, $O(n^2)$, $O(n^3)$ are safe? Justify each with arithmetic.

1.16 State a loop invariant for the maximum-scan of **Listing 1.3** and use it to prove the program correct.

1.21 Solutions to the Exercises

Solution 1.1. The program prints 1000. Here $n = 1$, so the horizontal constraint demands only 1 color, while the single column of 1000 cubes demands 1000 by the vertical constraint. The answer is $\max(1,1000) = 1000$, and the vertical constraint is doing all the work. Run on the input `1` / `1000`, **Listing 1.8** actually printed `1000`.

Solution 1.2. The answer is 7. The tallest column is 1 cube, so the vertical constraint asks for 1 color; but every column reaches height 1, so the bottom row is an unbroken substring of all seven columns and the horizontal constraint asks for 7. Thus $\max(7,1) = 7$. Run on `7` / `1 1 1 1 1 1 1`, **Listing 1.8** actually printed `7`. As a third case, `3` / `2 9 2` printed `9`, where the tall middle column dominates.

Solution 1.3. Take a gap-free run at row r occupying columns $j, j+1, \dots, k$. Their color indices before the $+1$ are $(j+r) \bmod C, (j+1+r) \bmod C, \dots, (k+r) \bmod C$ — the remainders of $k - j + 1$ *consecutive* integers. Consecutive integers have distinct remainders modulo C provided there are at most C of them, because the remainder increases by exactly one each time and only wraps after C steps. The run has at most n columns, and $C \geq n$ by construction. So no two coincide.

Solution 1.4. The answer is 0. All three cities share coordinate 5, so $x_{\max} = x_{\min} = 5$ and $2(5 - 5) = 0$: Cossack Vus is already standing in all three cities and need not move. The formula applies unchanged. Run on `3` / `5 5 5`, **Listing 1.9** actually printed `0`; on `1` / `7` it also printed `0`.

Solution 1.5. Without the obligation to return, the optimal plan is to start at one extreme and walk to the other, giving $x_{\max} - x_{\min}$ — exactly half the round trip. It is a lower bound because the walk must at some time be at $x_{\min}$ and at some other time at $x_{\max}$, and any walk between those two points costs at least their separation; it is achieved by the straight sweep. He should start at either extreme, $x_{\min}$ or $x_{\max}$; starting strictly between them forces a reversal and costs more. A program printing both variants actually printed `round trip = 6, one way = 3` for cities at 1,4, and `round trip = 16, one way = 8` for cities at 10,2,7,5.

Solution 1.6. Start with $n = 36$. Operation `1 10` is a multiplication. gcd(36,10): $36 \bmod 10 = 6$, so gcd(10,6); $10 \bmod 6 = 4$, so gcd(6,4); $6 \bmod 4 = 2$, so gcd(4,2); $4 \bmod 2 = 0$, answer 2. Then $n = 36/2 = 18$. **Print 18.** Operation `0 7` is an addition: a bijection, n unchanged. **Print 18.** Operation `1 6`: gcd(18,6): $18 \bmod 6 = 0$, answer 6. Then $n = 18/6 = 3$. **Print 3.** Operation `1 4`: gcd(3,4): $3 \bmod 4 = 3$, so gcd(4,3); $4 \bmod 3 = 1$, so gcd(3,1); $3 \bmod 1 = 0$, answer 1. Then $n = 3/1 = 3$. **Print 3.** On this input **Listing 1.10** actually printed `18`, `18`, `3`, `3`.

Solution 1.7. Let $g(i) = (i + x) \bmod n$ on the set $S = \{0,1,\dots,n-1\}$. Suppose $a, b \in S$ with $g(a) = g(b)$. Then $a + x \equiv b + x \pmod{n}$, so n divides $(a + x) - (b + x) = a - b$. But a and b both lie in $\{0,\dots,n-1\}$, so $|a - b| \leq n - 1 < n$; the only multiple of n of absolute value less than n is zero, hence $a = b$. So g is injective. An injective map from a finite set to itself is surjective, so g is a bijection and $|g(S)| = |S|$: the count of distinct values is unchanged. The argument used nothing about x, so it holds for every $x \geq 0$.

Solution 1.8. Multiplying every number by 0 makes them all 0, so exactly 1 distinct value should remain. By the standard convention $\gcd(n, 0) = n$ — every number divides 0, so the greatest common divisor of n and 0 is n itself — and the formula gives $n \leftarrow n/n = 1$. It agrees. The count can never rise again: an addition is a bijection on a one-element set, and a multiplication maps the single value 0 to 0. Run on `30` / `2` / `1 0` / `0 5`, **Listing 1.10** actually printed `1` then `1`.

Solution 1.9. $N = 4$, $K = 4$. *Minimum:* lead with 1; zeros to place $= N - K + 1 = 4 - 4 + 1 = 1$; remaining digits $= K - 2 = 2$, namely 2 and 3. That gives 1 0 2 3 = **1023**, digits {0,1,2,3}, four distinct. *Maximum:* nines to place $= 1$; remaining digits $= K - 1 = 3$, namely 8,7,6. That gives 9 8 7 6 = **9876**, digits {6,7,8,9}, four distinct. **Listing 1.11** actually printed `1023 9876`.

Solution 1.10. With $K = 1$ only one digit may be used, repeated eighteen times, and it cannot be 0 because the leading digit would then be zero. Smallest: eighteen 1s. Largest: eighteen 9s. **Listing 1.11** on `18 1` actually printed `111111111111111111 999999999999999999`.

Solution 1.11. The formula is popcount$(m-1)$ mod 3, so we need $p = 99$. Convert: $99 - 64 = 35$; $35 - 32 = 3$; $3 - 2 = 1$; $1 - 1 = 0$. So $99 = 64 + 32 + 2 + 1$, which in binary is 1100011. The number of 1-bits is 4, and 4 mod 3 = **1**. **Listing 1.13** on $m = 100$ actually printed `1`. Two more checks: $m = 64$ gives $p = 63 = 111111_2$, popcount 6, 6 mod 3 = 0, and the program printed `0`; $m = 1024$ gives $p = 1023$, ten 1-bits, 10 mod 3 = 1, and the program printed `1`.

Solution 1.12. Induction on the stage. The stage-0 sequence is the single digit 0, which lies in $\{0,1,2\}$. If every digit of stage k lies in $\{0,1,2\}$, then stage $k+1$ is stage k followed by a copy in which each value v is replaced by $(v+1)$ mod 3. The operation mod 3 returns a value in $\{0,1,2\}$ by definition of a remainder on division by 3. So every digit of stage $k+1$ lies in $\{0,1,2\}$ too, and the claim holds for all stages.

Solution 1.13. $f(10) = f(5) + f(3)$. Descend into $f(5) = f(2) + f(1)$; descend into $f(2) = f(1) + f(0)$; descend into $f(1) = f(0) + f(0) = 1 + 1 = 2$. **Cache** $f(1) = 2$. $f(2) = 2 + 1 = 3$. **Cache** $f(2) = 3$. $f(5) = 3 + 2 = 5$, the second term from the cache. **Cache** $f(5) = 5$. $f(3) = f(1) + f(1) = 2 + 2 = 4$, both from the cache. **Cache** $f(3) = 4$. $f(10) = 5 + 4 =$ **9**. Cache order: $f(1), f(2), f(5), f(3), f(10)$, with $f(0)$ seeded before the run. **Listing 1.15** on input `10` actually printed `9`.

Solution 1.14. An array must be allocated over the whole index range, and the index here can be as large as 10^{18}. At eight bytes per `long long` that is 8×10^{18} bytes — eight million terabytes — against a memory limit of 128 megabytes. A `map` stores only the keys actually visited, which **Listing 1.14** measured as 1,012 for the worst case: about sixteen kilobytes of payload. The ratio between what an array would need and what the map needs is roughly 5×10^{14}.

Solution 1.15. Take the working budget of 10^8 operations per second. $O(n)$: 10^5 operations. Safe by three orders of magnitude. $O(n\log n)$: $\log_2 10^5 \approx 16.6$, so about 1.7×10^6 operations. Safe. $O(n^2)$: $(10^5)^2 = 10^{10}$ operations, one hundred times the budget. Not safe — expect roughly one hundred seconds. $O(n^3)$: 10^{15} operations, ten million times the budget. Hopeless. So $O(n)$ and $O(n\log n)$ are safe; $O(n^2)$ and $O(n^3)$ are not.

Solution 1.16. *Invariant:* at the start of the pass with counter i, the variable `best` holds the maximum of $a[0 \ldots i-1]$. *Initialization:* before the loop, $i = 1$ and `best` $= a[0]$, which is the maximum of the single-element range $a[0 \ldots 0]$. *Maintenance:* assume the invariant at the start of a pass. The body sets `best` to $a[i]$ if $a[i] >$ `best`, and otherwise leaves it. Either way, afterward `best` equals $\max(\max(a[0..i-1]), a[i]) = \max(a[0..i])$. Incrementing i makes this the invariant for the next pass. *Termination:* the loop ends with $i = n$, so `best` is the maximum of $a[0 \ldots n-1]$, the whole array — which is what the program prints. On the array [3,9,2,9,4,1], **Listing 1.3** actually printed `maximum = 9`. ■

References

[1] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[2] D. Gries, *The Science of Programming*. New York, NY, USA: Springer-Verlag, 1981.

[3] E-Olymp (Eolymp Basecamp) Online Judge. Accessed: Sep. 24, 2026. [Online]. Available: https://eolymp.com/en/problems

[4] Sphere Online Judge (SPOJ). Accessed: Sep. 19, 2026. [Online]. Available: https://www.spoj.com

[5] "UVa Online Judge," *Wikipedia*. Accessed: Sep. 24, 2026. [Online]. Available: https://en.wikipedia.org/wiki/UVa_Online_Judge

[6] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, 4th ed. Morrisville, NC, USA: Lulu Press, 2020.

[7] beecrowd, community page. Accessed: early 2026. [Online]. Available: https://beecrowd.com

[8] UVa Online Judge. Accessed: Sep. 24, 2026. [Online]. Available: https://onlinejudge.org

[9] R. Soelaiman, S. Candra, and I. Subakti, "Efficient approach for deterministic data extrapolation from a clean periodic function with periodic components representation by a system of linear equations," *Journal of Theoretical and Applied Information Technology*, vol. 97, no. 8, pp. 2225–2239, Apr. 2019.

[10] R. Soelaiman, Y. H. Pratama, I. Subakti, and Y. Purwananto, "Genetic algorithm approach for automated generation of neural networks architecture for robust digit recognition," *Journal of Theoretical and Applied Information Technology*, vol. 98, no. 17, pp. 3479–3492, Sep. 2020.

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

[12] "Introduction to Algorithms," *Wikipedia*. Accessed: Sep. 24, 2026. [Online]. Available: https://en.wikipedia.org/wiki/Introduction_to_Algorithms

[13] S. Halim, F. Halim, and S. Effendy, *Competitive Programming Book* (official site). Accessed: Sep. 24, 2026. [Online]. Available: https://cpbook.net

[14] "GNU Compiler Collection," *Wikipedia*. Accessed: Sep. 24, 2026. [Online]. Available: https://en.wikipedia.org/wiki/GNU_Compiler_Collection

[15] "Clang," *Wikipedia*. Accessed: Sep. 24, 2026. [Online]. Available: https://en.wikipedia.org/wiki/Clang

[16] Republic of Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, Jakarta, Indonesia, 2022.

Chapter 2

Getting Started: the RAM Model, Insertion Sort, and Searching

By the end of this chapter you will be able to say exactly what an algorithm is and how it differs from a program; count the running time of a piece of code by hand, line by line, without running it; prove that a loop does what you claim it does, using a loop invariant; derive from first principles why Insertion Sort takes time proportional to n^2 in the worst case and only n in the best; explain why an $O(n\lg n)$ algorithm finishes in a blink where an $O(n^2)$ algorithm runs for a quarter of an hour; and write, trace and analyze both linear search and binary search in C++, including the two off-by-one traps that catch almost every beginner.

2.1 Where We Left Off

Chapter 1 was a chapter of *doing*. We took six real problems from the E-Olymp and SPOJ judges — Columns, Journey, Unique Numbers, Minimum and Maximum Numbers, Interesting Sequence, and Recursive Function 1 — and we solved all six. We introduced the Five-Phase Framework: read the problem until you can restate it in one sentence; find the structure; choose the paradigm; estimate the cost before you write code; then implement and test. We wrote C++ that compiled, ran, and was accepted by the judge.

What we did *not* do was justify any of it.

We said "this solution is $O(n)$, so it will pass" without ever defining what $O(n)$ means. We said Euclid's method needs only $O(\log\min(a,b))$ steps, and that memoizing a recursion turns something hopeless into something instant, in a notation we had not yet defined. In Chapter 1 those statements were promises. This chapter is where we begin to pay them, and Chapter 3 completes the payment.

The plan is simple and it runs in one direction. First we agree on words: what a *problem* is, what an *instance* of a problem is, what an *algorithm* is, and what a *program* is. These four are not synonyms and confusing them is the single most common source of muddled thinking in this subject. Then we agree on a *machine* — an imaginary, idealized computer called the RAM model — because you cannot count steps until you have agreed what one step is. Then we take one algorithm, Insertion Sort, and we do everything to it: describe it, trace it by hand, prove it correct, count its operations, and derive its running time exactly. Then we do the same, more quickly, for the two searching algorithms. By the end you will own a method that you can apply to any algorithm in the rest of the book.

> ✓ **KEY INSIGHT.** Chapter 1 gave you the *habit* of solving problems. Chapter 2 gives you the *vocabulary and the arithmetic* to say why a solution is good. The habit without the arithmetic is guesswork; the arithmetic without the habit is sterile. You need both.

2.2 What This Book Is Really About

At its core, computer science is about one thing: solving problems. Not just any problems — problems that arise in the real world and can be modeled computationally. Here are five, drawn from five different industries:

- Getting directions from your home in Keputih to your office in Rungkut. This is the **shortest path** problem, and we will solve it properly in Chapter 13.
- Scheduling airline flights so that no two aircraft are assigned the same gate at the same time. This is a **constraint satisfaction** problem.
- Searching through billions of web pages to find the ones that answer your question. This is **information retrieval**.
- Simulating how air flows over a jet engine blade. This is **numerical computation**.
- Compressing a video so it can be streamed over a phone connection. This is **data compression**, and the Huffman coding you meet in Chapter 9 is one of its oldest tools.

This book teaches you how to take such problems and solve them *well* — with algorithms that are both **correct**, meaning they give the right answer, and **efficient**, meaning they give it fast.

2.2.1 What this book IS about

There are exactly three activities, and everything in the sixteen chapters is one of them.

Designing algorithms. Inventing the step-by-step strategy that turns the input into the output. This is the creative act. There is no mechanical procedure for it — that is why we study paradigms, which are families of strategies that have worked before.

Analyzing algorithms. Proving that the strategy is correct, and measuring how fast it runs. This is the mathematical act, and it is the part most students skip and later regret skipping.

Choosing data structures. Organizing the data so that the algorithm can do its work cheaply. A brilliant algorithm on the wrong data structure is a slow program.

2.2.2 What this book is NOT about

It is just as important to say what we are *not* doing, so that you know when you are off track.

Programming languages. We write in C++ throughout this book, but we do not study C++. We will not discuss templates for their own sake, or the subtleties of move semantics. C++ is our notation, the way LaTeX is a mathematician's notation.

Computer architecture. We deliberately abstract the hardware away, using the RAM model of Section 2.6. Caches, pipelines and branch predictors are real and they matter in industry, but they would drown the ideas we are here to learn.

Software engineering. Large-scale system design, testing frameworks, version control, documentation practice — all essential, all taught elsewhere.

Complexity theory in depth. We touch computational complexity — the study of what is inherently hard — but our focus is on designing algorithms that work, not on classifying problems that do not have any.

Our focus is the **algorithm**: the pure, abstract idea of how to solve a problem, independent of what language or machine implements it.

2.3 A Brief History of Algorithms

Algorithms are much older than computers. Understanding that fact is the fastest way to stop thinking of an algorithm as "some code" and start thinking of it as an idea.

2.3.1 Where the word comes from

The word *algorithm* comes from a person's name. Muhammad ibn Musa **al-Khwarizmi** (circa 780–850 CE) was a Persian mathematician, astronomer and geographer working in the House of Wisdom in Baghdad. He wrote a book on solving equations whose title gave us the word *algebra*, and another on Hindu–Arabic numerals explaining, step by step, how to add, subtract, multiply and divide using the decimal positional system that had recently arrived from India. When that second book was translated into Latin in the twelfth century, his name was Latinized to *Algorismus*, and for centuries "algorism" meant the decimal calculation procedures he had described. Over time the word drifted, in both spelling and meaning, to the modern *algorithm*: any precise procedure for computing something.

There is a lesson hiding in that history. Al-Khwarizmi's great contribution was not a theorem. It was a *procedure*, written down so clearly that a person who did not understand why it worked could still carry it out correctly. That is exactly what an algorithm is, and it is exactly what you are learning to write.

As **Figure 2.1** shows, we are borrowing a name that is more than a thousand years old.

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

2.3.2 The first algorithm

If the word is a thousand years old, the thing itself is far older. The oldest non-trivial algorithm still in daily use is the **Euclidean algorithm** for the greatest common divisor of two integers, written down in Euclid's *Elements* around 300 BCE. You already used it in Chapter 1, in the Unique Numbers problem. It is over 2,300 years old and it is still, for its purpose, essentially the best method known.

Here it is, in one line: to find $\gcd(a, b)$, replace the pair (a, b) by the pair $(b, a \bmod b)$ and repeat until the second number is zero; the first number is then the answer. Let us run it on gcd(48,18), one step per line:

1. $a = 48$, $b = 18$. Since $b \neq 0$, compute $48 \bmod 18 = 12$, and replace the pair by (18,12).
2. $a = 18$, $b = 12$. Since $b \neq 0$, compute $18 \bmod 12 = 6$, and replace the pair by (12,6).
3. $a = 12$, $b = 6$. Since $b \neq 0$, compute $12 \bmod 6 = 0$, and replace the pair by (6,0).
4. $a = 6$, $b = 0$. Since $b = 0$, stop and report 6.

And indeed $\gcd(48,18) = 6$. Four lines of arithmetic, no computer, no programming language. That is an algorithm. Euclid never saw a machine, and his procedure runs unchanged on every machine ever built.

2.3.3 The nineteenth century: machines and the first programmer

Charles Babbage (1791–1871) designed, but never finished building, the **Analytical Engine** — the first design for a general-purpose, programmable computing machine. It had a "mill" (what we would call a processor), a "store" (memory), and was to be programmed with punched cards borrowed from the Jacquard loom. **Figure 2.2** shows him.

Figure 2.2 Charles Babbage (1791–1871), designer of the Analytical Engine, the first general-purpose computing machine ever conceived.

Ada Lovelace (1815–1852), working from Babbage's designs, wrote a detailed method for the Analytical Engine to compute the Bernoulli numbers. That method is generally considered the first algorithm ever written specifically to be executed by a machine, which makes her, by common consent, the first programmer. She also saw further than Babbage: she wrote that the Engine might one day manipulate not only numbers but any symbols whose relations could be expressed — music, for example. That is a remarkably modern thought for 1843. She is shown in **Figure 2.3**.

Figure 2.3 Ada Lovelace (1815–1852), who wrote the first algorithm intended to be executed by a machine.

2.3.4 The twentieth century: what computation *is*

Three people turned computing from engineering into a science.

Alan Turing (1912–1954), in a 1936 paper, defined an imaginary machine — a tape, a head that reads and writes symbols, and a finite table of rules — and argued that anything a human "computer" could calculate by following rules, this machine could calculate too. The **Turing machine** gave us, for the first time, a precise mathematical definition of what it means for something to be computable, and with it the proof that some perfectly well-posed problems have *no* algorithm at all. See **Figure 2.4**.

Figure 2.4 Alan Turing (1912–1954), who formalized computation with the Turing machine in 1936.

Alonzo Church (1903–1995) arrived at the same frontier from a different direction, with the **lambda calculus**: a notation in which everything is a function, and computation is the repeated substitution of arguments into function bodies. Church's system and Turing's machine turned out to define exactly the same class of computable functions, a coincidence so striking that it is now called the Church–Turing thesis. Every functional programming language alive today is a descendant of Church's notation. He is shown in **Figure 2.5**.

Figure 2.5 Alonzo Church (1903–1995), inventor of the lambda calculus, who shares his name with Turing in the Church–Turing thesis.

John von Neumann (1903–1957) did something more practical: he described, in a 1945 report, an architecture in which the program is stored in the same memory as the data it operates on. That single decision — the **stored-program computer** — is why your laptop can run a program it has never seen before simply by loading it into RAM. Virtually every computer in the world is still a von Neumann machine, and the RAM model we are about to adopt is a direct idealization of his design. He is shown in **Figure 2.6**.

Figure 2.6 John von Neumann (1903–1957), whose stored-program architecture underlies essentially every computer in use today.

> ✓ **KEY INSIGHT.** Algorithms are older than computers. The ideas are timeless — what changes is the hardware that runs them. Euclid's algorithm has outlived the Greek city-state, the Roman Empire, the vacuum tube and the floppy disk. The C++ you write this week will be obsolete in twenty years; the algorithm it implements may not be.

2.4 The Vocabulary: Problem, Instance, Algorithm, Program, Data Structure

This section defines five words. They are used loosely in conversation and precisely in this book, and almost every confusion a beginner has about algorithm analysis can be traced to blurring two of them together. Read this section slowly.

2.4.1 An algorithmic problem

> **DEFINITION.** An **algorithmic problem** is a specification of a desired relationship between input and output. It says *what* must be computed. It does not say *how*.

A problem specification has exactly two halves. The first half says what counts as a legal input. The second half says, for each legal input, what counts as a correct output.

Here is the problem we will spend most of this chapter on.

> **EXAMPLE — the SORTING problem.**
>
> *Input:* a sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.
>
> *Output:* a permutation (a reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of that same sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Notice three things about that specification. It never mentions a programming language. It never mentions a method. And it covers *infinitely many* different inputs at once: every finite sequence of numbers that has ever existed or ever will.

The symbol n there is our first piece of notation, and it will follow us through the whole book. It stands for the **size of the input** — here, how many numbers there are. The angle brackets $\langle \cdots \rangle$

simply mean "a sequence, in this order"; unlike a set, a sequence may repeat values and the order matters. The prime marks in a'_1 are read "a-one-prime" and just mean "the first element of the output", which is generally not the same as a_1, the first element of the input.

2.4.2 An instance of a problem

> **DEFINITION.** An **instance** of a problem is one particular legal input to it, together with everything needed to make the question concrete.

The SORTING problem is a problem. The sequence ⟨5,2,4,6,1,3⟩ is an *instance* of it. The correct output for that instance is ⟨1,2,3,4,5,6⟩.

Why does this distinction matter enough to deserve a definition? Because *correctness is a statement about the problem, and running time is usually a statement about the instance.* An algorithm is correct only if it produces the right answer on every instance, without exception. But it may be lightning-fast on one instance and hopelessly slow on another of exactly the same size. We will see precisely this with Insertion Sort in Section 2.11: on the instance ⟨1,2,3,4,5,6⟩ it does almost nothing, and on the instance ⟨6,5,4,3,2,1⟩ — same size, same numbers — it does the maximum possible work.

2.4.3 An algorithm

> **DEFINITION.** An **algorithm** is a well-defined computational procedure that takes some value, or set of values, as **input**, and produces some value, or set of values, as **output**. It is a sequence of computational steps that transforms the input into the output.

An algorithm is the *solution*, where the problem was the *question*. And for any problem there are infinitely many correct algorithms — some fast, some slow, some elegant, some ugly. The whole challenge of this book is to find, among that infinity, the efficient ones.

To count as an algorithm at all, a procedure must have five properties. These are the classical criteria, due to Knuth, and you should be able to test any procedure against them.

1. **Input.** Zero or more values are supplied from outside. (Zero is allowed: "print the first thousand prime numbers" takes no input.)
2. **Output.** At least one value is produced. A procedure that computes nothing observable is not solving anything.
3. **Definiteness.** Every step is precisely and unambiguously specified. "Stir until it looks right" is not definite. "Compare $A[i]$ with key and if $A[i]$ is larger, copy $A[i]$ into $A[i+1]$" is.
4. **Finiteness.** The procedure terminates after a finite number of steps, for *every* legal input. This is the property that separates an algorithm from a mere procedure: `while (true) i = i + 1;` is definite, effective, and useless.
5. **Effectiveness.** Every step is basic enough that a person could in principle carry it out exactly, with pencil and paper, in finite time. "Decide whether this arbitrary program halts" is not effective — Turing proved no such step exists.

> **EXAMPLE — testing a procedure against the five properties.** Consider: *"To sort a list, generate every possible ordering of it in turn; stop when you generate one that is sorted."* Input: yes, the list. Output: yes, the sorted list. Definiteness: yes, if we fix an order in which to generate permutations. Finiteness: yes — there are only $n!$ orderings and at least one is sorted, so we must stop. Effectiveness: yes, a person could do it. So it *is* an algorithm. It is also a catastrophically bad one: for $n = 20$ it may generate $20! \approx 2.4 \times 10^{18}$ orderings. Being an algorithm is a low bar. Being a *good* algorithm is the bar we care about.

2.4.4 A program

DEFINITION. A **program** is an implementation of an algorithm in a specific programming language, expressed so that a particular machine can execute it.

The algorithm is the abstract idea; the program is the concrete realization. One algorithm can be implemented by countless different programs — in C++, in Python, by one student cleanly and by another badly — and they are all the same algorithm. This is why we analyze algorithms rather than programs: the analysis then applies to every implementation at once, on every machine, forever. **Figure 2.7** pictures the relationship: one algorithm, many programs, many machines — and one analysis that covers them all.

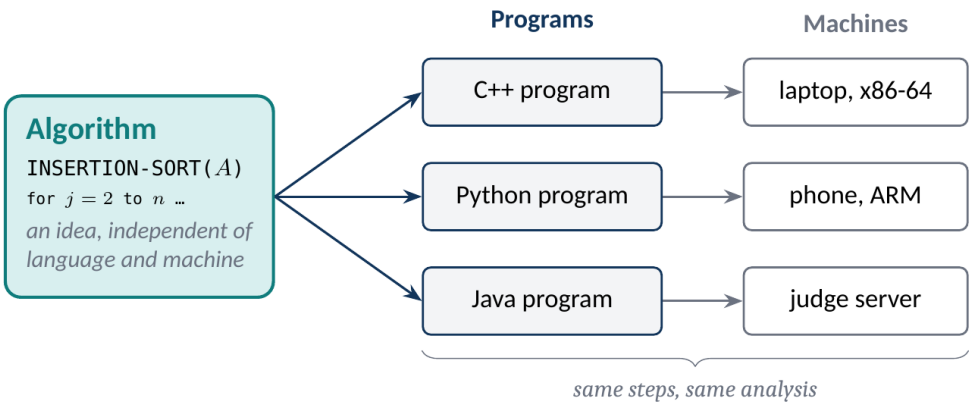


Figure 2.7 The algorithm is an idea; the program is what that idea becomes on a particular machine, in a particular language.

2.4.5 A data structure

DEFINITION. A **data structure** is a particular way of organizing and storing data in a computer so that it can be accessed and modified efficiently.

Arrays, linked lists, stacks, queues, hash tables, heaps, binary search trees — each stores the same kind of information but makes different operations cheap and different operations expensive. An array makes "give me the element at position 7" instantaneous but "insert something at position 7" expensive. A linked list does the reverse.

Analogy. If the algorithm is the recipe — "sort these numbers" — then the data structure is the kitchen equipment. The same recipe is quick with a food processor and slow with a mortar and pestle. Better equipment does not change the recipe; it changes how long the recipe takes. Choosing the data structure *is* part of designing the algorithm, which is why the two subjects are always taught together.

2.4.6 Correctness and efficiency, defined

Two more definitions, and then we have all our words.

DEFINITION. An algorithm is **correct** for a problem if, for **every** legal instance of that problem, it halts and produces an output satisfying the problem's output specification. An algorithm that gives the right answer on every instance but one is not "almost correct"; it is incorrect.

DEFINITION. The **efficiency** of an algorithm is the amount of resource it consumes as a function of the input size n — chiefly *time*, measured as a count of primitive steps, and secondarily *space*, measured as a count of memory cells.

The phrase *as a function of the input size* is the heart of the matter and is what the rest of this chapter is about. We do not ask "how long does this take?"; we ask "how does the time **grow** when n grows?" That question has a machine-independent answer, and the machine-specific one does not.

> ⚠ **THINK.** Among the infinitely many correct algorithms for a given problem, find the ones that are EFFICIENT. That is the whole book in one sentence. Correctness gets you an answer; efficiency gets you an answer before the deadline, before the battery dies, before the customer leaves.

2.5 The Five Design Goals

When we build algorithms and data structures we are aiming at five goals, and it helps to know from the start which two this book is about and which three it is not. **Figure 2.8** groups them: the first two belong to the *design* of the data structure and algorithm, the last three to the *implementation*.

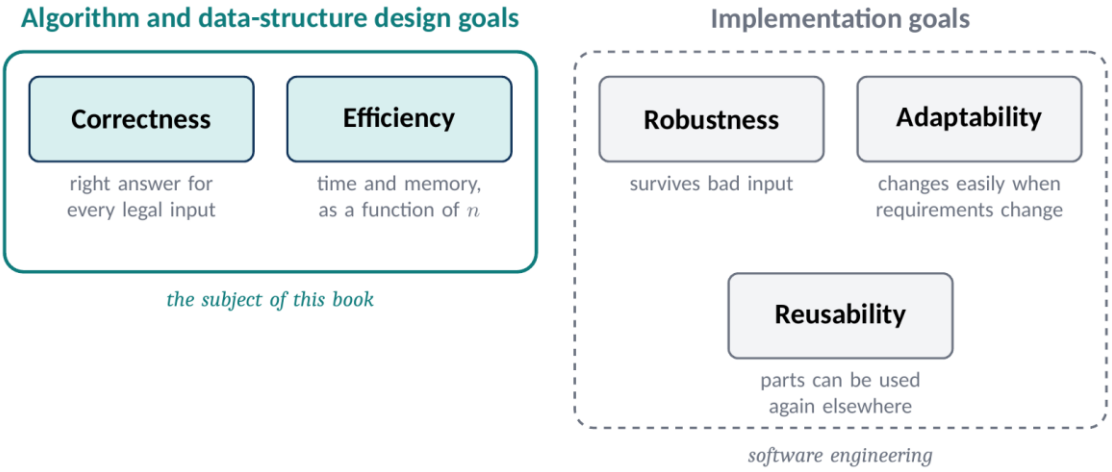


Figure 2.8 The five goals, in two groups. Correctness and efficiency are design goals and are the subject of this book; robustness, adaptability and reusability are implementation goals belonging to software engineering.

Correctness (★ primary). The algorithm produces the correct output for *every* valid input — not for the three test cases you happened to try. Testing can only ever show the presence of bugs, never their absence, so we prove correctness rather than test for it. The tools are loop invariants, which we meet in Section 2.9, and mathematical induction, which is the same tool wearing different clothes.

Efficiency (★ primary). The algorithm uses as little time and memory as possible. We measure this by asymptotic analysis, which tells us how the running time grows as the input size grows. Chapter 3 makes this precise with the O, Θ and Ω notations; this chapter gets you to the point where you want them.

Robustness. The program handles erroneous or unexpected input gracefully instead of crashing or returning nonsense. If a user of a sorting routine hands it a null pointer, or an array length of -1, what happens? That is a vital question — and it is a software engineering question, because it concerns the program, not the algorithm.

Adaptability. The program can be modified easily when the requirements change, as they always do. Good algorithm design helps here almost for free: a clean, well-structured solution built out of understood parts is far easier to change than a tangle.

Reusability. The code can be used again in another program. Standard algorithms — sorting, searching, graph traversal — are the ultimate reusable components, which is precisely why they are standardized in libraries like the C++ Standard Template Library. Every hour you spend understanding how `std::sort` works is an hour you never have to spend writing it.

> ✓ **KEY INSIGHT.** In this book we focus almost entirely on correctness and efficiency. The other three goals are important — they are what separates a working prototype from a shipped product — but they belong to your software engineering courses. Here, if an algorithm is correct and fast, it has done its job.

2.6 The RAM Model of Computation

> ❓ **QUESTION.** Before we can analyze how fast an algorithm runs, we must agree on what "one step" means. A modern processor executes some instructions in a single clock cycle and others in hundreds. Which is a step? How do we count?

The answer is that we stop trying to describe a real machine and adopt an agreed, idealized one. The model is called the **Random Access Machine**, or **RAM model**, and it is the model in which every running time in this book is stated.

2.6.1 The rules of the model

The RAM model is an imaginary computer with these properties.

1. It has a memory made of numbered cells, and it may read or write **any** cell in one time step, regardless of which cell it is. That is what *random access* means: reaching cell number 1,000,000 costs exactly what reaching cell number 3 costs.
2. Instructions are executed one after another, with no concurrency. There is one processor and it does one thing at a time.
3. Each **simple** instruction takes exactly **one** time step.
4. Data are integers and floating-point numbers of *reasonable* size — specifically, each value fits in a constant number of memory cells, and the number of bits in a word is at least $\lg n$ so that an index into the input can be held in one word. (The symbol $\lg n$ means $\log_2 n$, the logarithm to base 2; we will define it carefully in Section 2.12.)

2.6.2 What counts as one step

The instructions the model charges one step for fall into three families.

Arithmetic. Addition, subtraction, multiplication, division, remainder (`%`), floor, ceiling, and comparison of two numbers. Also the bitwise operations: `&`, `|`, `^`, `<<`, `>>`.

Data movement. Loading a value from memory into a register, storing a register back to memory, copying one variable into another. Crucially, **indexing an array**, `A[i]`, is one step: the machine computes the address and fetches the cell, and by rule 1 that costs the same wherever the cell lives.

Control. Testing a condition in an `if` or a `while`; jumping to another instruction; calling a subroutine; returning from one.

So in the RAM model, each of these C++ lines costs exactly one step:

```
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
```

2.6.3 What does NOT count as one step

This is the part students skip, and it is where wrong analyses come from.

A loop is not one step. `for (int i = 0; i < n; ++i) x = x + 1;` contains only constant-cost lines, but it executes them n times. The cost is n steps, not one. You must always multiply the cost of a line by the number of times it runs — that is the whole technique of Section 2.7.

A function call is not one step unless the function itself is constant-cost. Calling `sort(v)` is one line of your program and perhaps $n\lg n$ steps of work. In an analysis you must open the box and count what is inside.

Operations on unbounded data are not one step. Multiplying two numbers with a million digits each is not a single operation; the RAM model's "reasonable size" clause excludes it. Likewise, copying a string of length m costs m, not 1, because it is m separate cell movements wearing one name.

Exponentiation by an arbitrary amount is not one step. Computing 2^k where k can grow with n takes about $\lg k$ multiplications by repeated squaring; charging it one step would let you smuggle enormous work past the accountant.

> ⚠ **THINK.** When in doubt, ask: *does the amount of work this line does depend on n?* If the answer is no, it is one step. If the answer is yes, it is not one step, and you must count what is inside it.

2.6.4 Why choose this model?

The RAM model is, plainly, a lie. Real computers have caches that make some memory accesses fifty times slower than others; they have instruction pipelines, branch prediction, multiple cores, and virtual memory. Rule 1 in particular — that every memory access costs the same — is exactly the assumption that modern hardware works hardest to violate.

So why use it?

Because it is machine-independent. If we measured wall-clock seconds, the same algorithm would appear to get faster every time someone bought a new laptop, and two students measuring the same code would report different results. The RAM model counts *operations*, not seconds. An operation count is a property of the algorithm and the input, not of the shop you bought your computer from.

Because it is predictive. This is the real justification, and it is empirical rather than theoretical: algorithms that the RAM model says are faster generally *are* faster on real hardware, once n is large. The model gets the ranking right even though it gets the constants wrong. You will see this demonstrated with a stopwatch in Section 2.11.4.

Because it is simple enough to compute with. A model that faithfully described cache hierarchies would be so complicated that nobody could analyze anything with it. There are such models — the external-memory model, the cache-oblivious model — and they are used when memory traffic genuinely dominates. For everything in this book, RAM is the right level of detail.

✓ **KEY INSIGHT.** A model is not a description of reality; it is a decision about what to ignore. The RAM model ignores everything about the hardware except that memory is uniform and work is sequential. It keeps exactly what is needed to compare algorithms fairly, and throws away exactly what would otherwise make the comparison depend on your budget.

2.7 How to Analyze Running Time

We now have a machine. Here is the method for counting on it.

DEFINITION. The **running time** of an algorithm on a particular input is the total number of primitive steps executed in the RAM model. We write it $T(n)$, read "T of n", where n is the size of the input. It is a **count of operations**, never a number of seconds.

The procedure has three parts, and it never varies:

1. **Assign a cost to each line.** In the RAM model every line of straight-line code costs some constant. We give the constants names: c_1 for the first line, c_2 for the second, and so on. We do not try to find their numerical values — they depend on the compiler and the processor, and, as we shall see, they will not matter.
2. **Count how many times each line executes.** This is the part that requires thought, because it depends on the input.
3. **Sum cost × times over all lines.** The result is $T(n)$.

That is all there is to it. The skill is entirely in step 2.

2.7.1 What is the input size?

The symbol n has to be defined for each problem, because "size" means different things in different settings:

- **Sorting and searching:** n is the number of elements in the array. This is the case for the whole of this chapter.
- **Matrix operations:** n is the dimension of the matrix, so the input actually contains n^2 numbers. An algorithm running in time n^3 is therefore only about $n^{1.5}$ times the size of its input — worth remembering before you call it slow.
- **Graph algorithms:** two parameters are needed, n (or $|V|$) for the number of vertices and m (or $|E|$) for the number of edges. From Chapter 10 onward all our running times mention both.
- **Number-theoretic problems:** n is the number of **bits** in the input number, not the number itself. Testing whether a 2,048-bit number is prime is a problem of size 2,048, not of size 2^{2048}. Get this wrong and every complexity claim in cryptography will look absurd to you.

⚠ **THINK.** Always state what n means before you use it. "This runs in $O(n)$" is meaningless until the reader knows whether n counts elements, vertices, or bits.

2.7.2 Running time is not wall-clock time

It bears repeating, because it is the most common misunderstanding in the subject. Running time, in this book, is the number of primitive operations executed. It is not the number of seconds elapsed. The two are related — on any one machine, seconds are roughly proportional to operations — but only the operation count is a property of the algorithm.

If you time a program and get 0.8 seconds, you have learned something about your laptop, your compiler flags, your operating system's mood, and, somewhere in there, your algorithm. If you

count that it executes about $n^2/2$ operations, you have learned something that will still be true in fifty years.

2.7.3 A first complete example

Let us do the whole procedure once on something tiny, so the method is clear before the data gets complicated. **Pseudocode 2.1** finds the largest value in an array of n numbers:

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
```

Now the three steps.

Step 1, costs. Line 1 is one array access and one store: cost c_1. Line 2's loop machinery — test $i \leq n$, increment i — costs c_2 per test. Line 3 is one array access and one comparison: c_3. Line 4 is one access and one store: c_4. Line 5 is a return: c_5.

Step 2, counts. Line 1 runs once. Line 2's test runs n times in total: once for each of $i = 2,3, \dots, n$, which is $n - 1$ successful tests, plus one final test with $i = n + 1$ that fails and ends the loop. Line 3 runs once per successful test, so $n - 1$ times. Line 4 runs only when the test on line 3 succeeds; call that number k, where $0 \leq k \leq n - 1$. Line 5 runs once.

Step 3, sum.

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

Since k is somewhere between 0 and $n - 1$, we can bound Equation (2.1) from both sides:

$$c_1 + c_2 n + c_3(n - 1) + c_5 \;\leq\; T(n) \;\leq\; c_1 + c_2 n + (c_3 + c_4)(n - 1) + c_5.$$

Both ends have the form $an + b$ for constants a and b. So whatever the input, $T(n)$ is a linear function of n. We will say in Chapter 3 that $T(n) = \Theta(n)$. Note what happened to k: it changed the constants but not the shape. That is the pattern you will see again and again.

2.8 Insertion Sort: the Algorithm

Insertion Sort is the first real algorithm we analyze completely, and we analyze it completely because it is the perfect teaching specimen: it is short enough to hold in your head, it is genuinely useful in practice for small or nearly-sorted arrays, its correctness proof is the textbook example of a loop invariant, and its best and worst cases differ dramatically, which forces us to confront the three-case analysis of Section 2.11.

The problem it solves is the SORTING problem of Section 2.4.1. **Figure 2.9** states that problem formally, with the input on the left, the output on the right, and — at the bottom — the two questions that this whole chapter exists to answer: is it correct, and how long does it take?

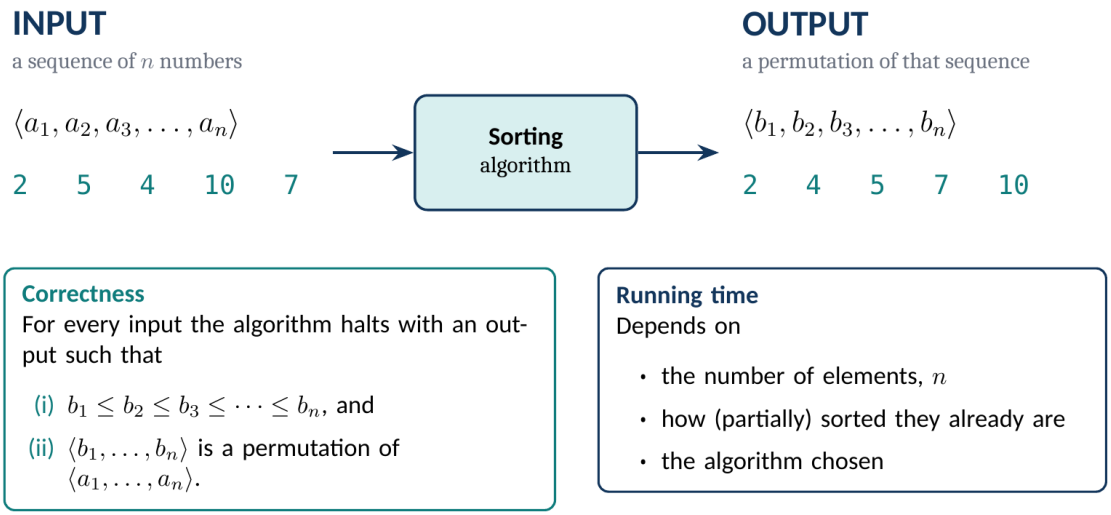


Figure 2.9 The sorting problem. The output $\langle b_1, \ldots, b_n \rangle$ must be both *ordered* ($b_1 \le b_2 \le \cdots \le b_n$) and a *permutation* of the input — two conditions, and an algorithm that meets only one of them is wrong. The running time depends on how many elements there are, how nearly sorted they already are, and which algorithm we chose.

Read the bottom-left box of that figure carefully, because it contains a trap. Correctness requires **two** conditions, not one. A procedure that outputs ⟨1,2,3,4,5⟩ for every input produces a perfectly ordered sequence and is completely wrong, because its output is not a permutation of the input. Ordering alone is not sorting.

2.8.1 The intuition: sorting a hand of cards

Imagine you are playing cards. You hold some cards in your left hand, and you keep them sorted at all times. The remaining cards are face down on the table. With your right hand you pick up the top card from the table, and you must place it into your left hand so that the hand stays sorted. How do you do it? You move the new card leftwards past every card that is bigger than it, and drop it into the gap. This is exactly **Figure 2.10**.

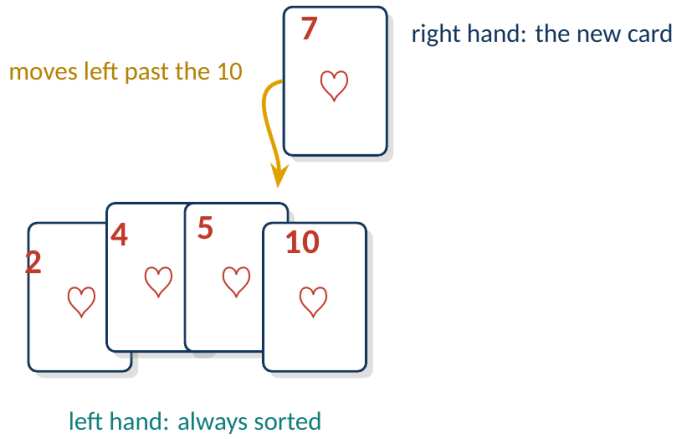


Figure 2.10 The card-player's method. The left hand holds a sorted run, 2–4–5–10; the right hand has just picked up the 7 and will move it leftwards past the 10 to reach its place.

Two features of that picture become two features of the algorithm.

First, **the sorted part grows by one at each step**. Before you pick up the 7, four cards are in order; after you place it, five are. This is the *incremental* method of Section 2.15.1: extend a correct solution for $j - 1$ items into a correct solution for j items.

Second, **you find the place by scanning right-to-left, not by searching**. You compare the new card with the rightmost card in your hand, then the next, and so on. You stop as soon as you meet a card that is not bigger than yours, because everything further left is smaller still — the hand is already sorted, so one failed comparison guarantees all the rest would fail too. That early stop is why the best case of Insertion Sort is so much better than its worst case.

2.8.2 The algorithm in pseudocode

Pseudocode 2.2 is the procedure. The array is called A, its length is `A.length`, and indices run from 1 to `A.length` — we count from 1, as CLRS [1] does, because it makes the mathematics cleaner. Section 2.8.4 explains how to get 1-based indices in C++, which counts from 0.

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
```

Walk through the seven lines in words, because every later section refers back to them.

Line 1 runs the outer loop. The variable j marks the card we are about to insert. We start at $j = 2$, not $j = 1$, because a one-element array $A[1..1]$ is already sorted and there is nothing to do.

Line 2 saves $A[j]$ into a variable called `key`. This save is essential: the shifting on line 5 is about to overwrite $A[j]$, and if we had not kept a copy we would lose the value we are trying to place. Beginners who omit line 2 and write `A[i+1] = A[j]` on line 7 produce an array full of duplicates.

Line 3 sets i to point at the rightmost element of the sorted part, $A[j-1]$.

Line 4 is the test of the inner loop. It has two conditions joined by `and`, and both matter. The condition $i > 0$ stops us walking off the left end of the array. The condition $A[i] > key$ stops us as soon as we meet an element that is not greater than the key — that is the early exit from Section 2.8.1. The order of the two conditions is not optional: in C++, `&&` evaluates left to right and stops at the first false, so writing `i > 0 && A[i] > key` guarantees we never read $A[0]$. Swap them and the program reads out of bounds.

Line 5 moves one element right, opening a gap. *Line 6* steps left. Together they are one iteration of the shift.

Line 7 drops the key into the gap. Why $A[i+1]$ and not $A[i]$? Because the loop exited *after* decrementing i past the last element it moved, so position i holds something that must stay to the left of the key, and position $i+1$ is the hole. This is the single most common off-by-one error in the algorithm, and Section 2.9 will prove that $i+1$ is right.

2.8.3 Tracing the algorithm by hand

Let us sort $A = \langle 5,2,4,6,1,3 \rangle$ completely by hand. **Table 2.1** traces it. Read it with the array in front of you, and check every line.

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]

Now let us expand the fifth row, $j = 5$, into individual operations, one per line, because that row is where all the work happens and a summary hides it.

At the start of this iteration the array is [2,4,5,6,1,3] and the first four entries are sorted.

1. Line 2: $key \leftarrow A[5] = 1$. The array is unchanged; we merely have a copy of the 1.
2. Line 3: $i \leftarrow 4$.
3. Line 4: test. Is $i > 0$? $4 > 0$, yes. Is $A[4] > key$? $A[4] = 6$ and $6 > 1$, yes. Enter the loop.
4. Line 5: $A[5] \leftarrow A[4]$, so the array becomes [2,4,5,6,6,3]. The 1 looks lost, but we saved it in `key`.
5. Line 6: $i \leftarrow 3$.
6. Line 4: test. $3 > 0$, yes. $A[3] = 5 > 1$, yes. Enter again.
7. Line 5: $A[4] \leftarrow A[3]$, giving [2,4,5,5,6,3].
8. Line 6: $i \leftarrow 2$.
9. Line 4: test. $2 > 0$, yes. $A[2] = 4 > 1$, yes. Enter again.
10. Line 5: $A[3] \leftarrow A[2]$, giving [2,4,4,5,6,3].
11. Line 6: $i \leftarrow 1$.
12. Line 4: test. $1 > 0$, yes. $A[1] = 2 > 1$, yes. Enter again.
13. Line 5: $A[2] \leftarrow A[1]$, giving [2,2,4,5,6,3].
14. Line 6: $i \leftarrow 0$.
15. Line 4: test. Is $i > 0$? $0 > 0$ is **false**. The loop ends. Note that we stopped on the *first* condition; had we tested $A[0]$ we would have read memory that does not belong to us.
16. Line 7: $A[i + 1] = A[1] \leftarrow key = 1$, giving [1,2,4,5,6,3].

Sixteen lines to move one card. The while-loop test on line 4 was evaluated five times: four times it succeeded and once it failed. Remember that number — five — because in Section 2.10 it is going to be called t_5, and the whole running-time analysis is built out of those numbers.

2.8.4 Insertion Sort in C++

Now the program. There is one decision to explain first. Our pseudocode indexes from 1; C++ indexes from 0. Three ways out exist: rewrite the algorithm for 0-based indices, translate mentally at every line, or declare the array one element longer and never use element 0. We take the third route throughout this chapter, because it lets the C++ sit line-for-line against the pseudocode, and a reader checking the program against the proof does not have to hold a translation in their head. In production code you would of course use 0-based indices; Exercise 2.3 asks you to make that change.

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

```
// 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;
}
```

Compiled with `g++ -std=c++17 -O2` and run, **Listing 2.1** prints:

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

That is the same answer we obtained by hand in **Table 2.1**, which is the least we should expect.

A program that only prints the answer, however, teaches nothing about *how* the answer arose. **Listing 2.2** adds a line of output per outer iteration, so the machine reproduces the hand trace of **Table 2.1**.

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

```
// 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;
}
```

Its actual output is:

```
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]
```

Compare that output line by line with **Table 2.1**. The `Result` column of the table and the bracketed array of the program agree on all five rows; the `shifts` count agrees with the number of "shift" words in the table's `Action` column — one, one, zero, four, three. The hand trace and the machine agree. That agreement is not a formality: whenever you invent an algorithm, trace it by hand *and* instrument it, and make the two match. Where they disagree, one of them is a bug, and it is usually worth knowing which.

2.9 Proving Insertion Sort Correct: the Loop Invariant

> **? QUESTION.** We have watched Insertion Sort work on one array of six numbers. How do we *prove* that it produces sorted output for **all** inputs — for every array, of every length, containing every combination of numbers?

You cannot answer that by testing. There are infinitely many inputs; you can try finitely many. The tool that closes the gap is the **loop invariant**.

2.9.1 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 begins, and true again immediately after the loop finishes.

The word "invariant" means *unchanging*: the truth of the statement does not change as the loop runs, even though the data it talks about changes constantly. A good invariant is a sentence that answers the question *"what has this loop achieved so far?"*

To use an invariant as a proof, you must establish three things.

1. **Initialization.** The invariant is true before the very first iteration begins.
2. **Maintenance.** *If* the invariant is true before some iteration, *then* it is still true before the next iteration. In other words, one trip around the loop preserves it.
3. **Termination.** When the loop finally stops, the invariant — combined with the reason the loop stopped — gives us the property we actually wanted to prove.

The first two steps are exactly mathematical induction. Initialization is the base case; maintenance is the inductive step. If the invariant holds at the start and each iteration preserves it, then by induction it holds at the start of iteration 1, iteration 2, iteration 3, and so on for however many iterations there are. Termination is the extra step that induction alone does not give you: it is where you cash in the invariant for a statement about the answer.

> **Why the third step is not optional.** It is easy to state an invariant that is true and useless. "At the start of each iteration, j is an integer" is perfectly invariant, and it tells you nothing about sorting. A useful invariant is one whose truth *at the moment the loop stops* is the thing you were trying to prove. Designing the invariant is the hard, creative part; verifying it is mechanical.

2.9.2 The invariant for Insertion Sort

Here it is.

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

Read it twice. It has two clauses, and both are needed, for exactly the reason **Figure 2.9** warned us about: the prefix must be **sorted**, and it must contain **the same elements** it started with. Drop the second clause and the invariant would be satisfied by an algorithm that overwrites the prefix with zeros.

Let us check the invariant against our trace. At the start of the iteration $j = 5$ in **Table 2.1**, the array was [2,4,5,6,1,3]. The subarray $A[1..4] = [2,4,5,6]$ is sorted: ✓. And $\{2,4,5,6\}$ is exactly the set of values that occupied those four positions in the original array [5,2,4,6, …], namely 5,2,4,6: ✓. Both clauses hold.

2.9.3 Initialization

We must show the invariant is true before the first iteration, which begins with $j = 2$.

Substituting $j = 2$ into the invariant, the claim is about the subarray $A[1 \dots 2-1] = A[1 \dots 1]$. That subarray contains exactly one element, $A[1]$.

Is it sorted? A sequence of one element is sorted, vacuously: sortedness requires $a_1 \le a_2 \le \cdots$, and with only one element there is no pair of adjacent elements to compare, so no condition can fail. Is it a rearrangement of the elements originally in $A[1..1]$? Yes — nothing has been executed yet, so $A[1]$ still holds precisely its original value.

Both clauses hold before the first iteration. ✓

A remark on vacuous truth. Beginners are often uneasy that "a one-element list is sorted" feels like cheating. It is not. The definition of *sorted* is "every adjacent pair is in non-decreasing order". In a list of one element there are zero adjacent pairs, and a statement about all members of an empty collection is automatically true — there is no counterexample available. This is the same reason "every unicorn in this room is purple" is true.

2.9.4 Maintenance

We must show that if the invariant holds at the start of an iteration, it holds at the start of the next one. So assume, for a particular value of j with $2 \leq j \leq n$, that $A[1 \ldots j-1]$ is a sorted rearrangement of its original contents. We must show that after the body of the loop runs, $A[1 \ldots j]$ is a sorted rearrangement of *its* original contents — because at the start of the next iteration the loop index is $j+1$ and the invariant then talks about $A[1 \ldots (j+1)-1] = A[1 \ldots j]$.

The body executes as follows. Line 2 copies $A[j]$ into `key`, so the value is safe. Line 3 sets $i = j - 1$. Then the inner `while` loop runs.

To reason about the inner loop we need an invariant for *it* as well. Loops nest, and so do their proofs.

INNER LOOP INVARIANT. At the start of each iteration of the `while` loop of line 4, the following are all true: (a) the positions $A[1 \ldots i]$ hold their values from the start of the outer iteration, unchanged; (b) the positions $A[i+2 \ldots j]$ hold the values that were originally in $A[i+1 \ldots j-1]$, each moved one place right; (c) position $A[i+1]$ is a *hole* — whatever it contains is a stale duplicate we are free to overwrite; and (d) every value in $A[i+2 \ldots j]$ is strictly greater than `key`.

Initialization of the inner loop. On entry, $i = j - 1$. Clause (a) says $A[1..j-1]$ is untouched — true, we have changed nothing. Clause (b) speaks of $A[j+1 \ldots j]$, an empty range, so it is vacuously true. Clause (c) says $A[j]$ is a hole, which is true because we copied its value into `key` and no longer need it there. Clause (d) speaks of the same empty range and is vacuous. ✓

Maintenance of the inner loop. Suppose the four clauses hold and the loop test succeeds, so $i > 0$ and $A[i] > key$. Line 5 copies $A[i]$ into the hole at $A[i+1]$; the value $A[i]$ now appears in position $i+1$, one place right of where it was, which extends clause (b) by one element, and since $A[i] > key$ that new element satisfies clause (d) too. Position i is now the stale duplicate, that is, the new hole. Line 6 decrements i, so clause (c) again names the hole correctly and clause (a) shrinks by exactly the one element we just moved. All four clauses hold again. ✓

Termination of the inner loop. The loop stops for one of two reasons.

Reason 1: $i = 0$. Then clause (d) says every element of $A[2 \ldots j]$ exceeds `key`, and clause (c) says $A[1]$ is the hole. Line 7 writes `key` into $A[1]$. The resulting $A[1 \ldots j]$ is `key` followed by $A[2..j]$, which by clauses (b) and the outer invariant is the old sorted $A[1..j-1]$ in its original order, shifted right. A sorted sequence, all of whose members exceed `key`, preceded by `key`, is sorted. ✓

Reason 2: $A[i] \leq key$ for some $i > 0$. Then clause (a) says $A[1 \ldots i]$ is the original sorted prefix's first i elements, so every one of them is $\leq A[i] \leq key$. Clause (d) says every element of $A[i+2 \ldots j]$ is $> key$. Line 7 writes `key` into the hole at $A[i+1]$. Reading left to right, we now have: elements all $\leq key$, then `key`, then elements all $> key$, with each of the two outer blocks internally sorted because they are contiguous pieces of the previously sorted prefix. The whole of $A[1 \ldots j]$ is therefore sorted. ✓

In both cases, line 7 writes to $A[i+1]$, and in both cases $A[i+1]$ is exactly the hole. That settles the off-by-one question raised in Section 2.8.2: the key goes at $i+1$, never at i, because line 6 always moves i one place past the last element it shifted.

Finally, no value was created or destroyed: every write either copied an element to a neighboring cell or wrote back the saved `key`, so $A[1 \dots j]$ is a permutation of its original contents. Both clauses of the outer invariant now hold for $j+1$. Maintenance is proved. ✓

2.9.5 Termination

The outer loop increments j by 1 each time and stops when j exceeds n, that is, when $j = n+1$. (It must stop: j starts at 2, strictly increases by 1 every iteration, and the test is against the fixed bound n, so after at most $n-1$ iterations the test fails. Insertion Sort satisfies the finiteness property of Section 2.4.3.)

Substitute $j = n+1$ into the invariant. It says: $A[1 \dots (n+1)-1] = A[1 \dots n]$ consists of the elements originally in $A[1 \dots n]$, in sorted order.

But $A[1 \dots n]$ is the entire array. So the whole array holds its original elements in sorted order. That is precisely the output specification of the SORTING problem — both clauses of it. Insertion Sort is correct. ■

> ✓ **KEY INSIGHT.** Loop invariants are the standard proof technique for iterative algorithms, and you will use them in every chapter from here to the end of the book. Think of an invariant as the answer to: *"What is always true about the state of the computation at this point?"* Write that sentence down before you write the loop, and the loop will usually come out right the first time.

> ⚠ **THINK.** Termination was the step that needed the least work here and it is the step that matters most. Notice that we had to combine *two* facts: the invariant, and the reason the loop stopped ($j = n+1$). The invariant alone never proves anything about the answer; it is the invariant *evaluated at the exit condition* that does.

2.10 Counting the Cost of Insertion Sort

Correctness is settled. Now: how long does it take?

We apply the method of Section 2.7 exactly. Assign a constant cost to each line, count how many times each line runs, and add up the products.

2.10.1 The cost-and-times table

There is one quantity we cannot pin down in advance. The inner `while` loop of line 4 runs a different number of times for each j, depending on the data. So we give it a name.

> **DEFINITION.** Let t_j denote the number of times the `while` loop **test** on line 4 is executed for that value of j.

Read that definition precisely, because everything downstream depends on it. It counts *tests*, not *successful* tests. For any given j the test is evaluated some number of times, all but the last of which succeed and send us into the body. So the body of the loop — lines 5 and 6 — executes $t_j - 1$ times. That single off-by-one is the reason the table below has t_j on one row and $t_j - 1$ on the next two.

Go back to the expanded trace of Section 2.8.3. For $j = 5$ we evaluated the test five times: four successes and one failure. So $t_5 = 5$, and lines 5 and 6 each ran $t_5 - 1 = 4$ times — which matches `shifts=4` in the program's output.

With t_j defined, here is the complete table, which we will now use to derive everything.

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	$n - 1$

Each count deserves a sentence of justification, because a count you cannot justify is a count you have guessed.

Line 1 runs n times. The loop header is evaluated once for each value $j = 2,3, \dots, n$ that enters the body — that is $n - 1$ evaluations — plus one final evaluation with $j = n + 1$ which fails and exits. Total n. This "one extra test to discover the loop is over" is a general fact about loops and it is where the n versus $n - 1$ difference between rows comes from.

Lines 2, 3 and 7 run $n - 1$ times. They sit inside the loop body, which is entered once for each j from 2 to n. That is $n - 1$ values.

Line 4 runs $\sum_{j=2}^{n} t_j$ times. For each j the test is evaluated t_j times by definition; sum over all j.

Lines 5 and 6 run $\sum_{j=2}^{n} (t_j - 1)$ times. For each j, the body is entered once for every *successful* test, which is one fewer than the number of tests.

The symbol $\sum$ is read "sum of". The expression $\sum_{j=2}^{n} t_j$ means: take t_2, then t_3, then t_4, and so on up to t_n, and add them all together. If $n = 6$ and the t values are 2,2,1,5,4, then $\sum_{j=2}^{6} t_j = 2 + 2 + 1 + 5 + 4 = 14$.

2.10.2 The general formula

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

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

Look at what Equation (2.2) contains. The constants $c_1, \dots, c_7$ depend on the compiler and the processor; we will never need their values. The quantity n is the input size. And the numbers $t_2, t_3, \dots, t_n$ depend on **which particular array we were given**. Two arrays of the same length can produce completely different sets of t_j values and therefore completely different running times.

> ✓ **KEY INSIGHT.** $T(n)$ is not determined by n alone. The formula above has a hole in it shaped like $\sum t_j$, and only the actual input can fill it. This is why the next section must split into three cases: we have to say *which* input we are talking about before we can finish the arithmetic.

2.11 Best Case, Worst Case, and Average Case

The same algorithm on the same number of elements can be fast or slow depending on the arrangement of those elements. **Figure 2.11** shows the shape of the situation: for each input size, the running time is not a single point but a band, and we summarize that band by its top edge, its bottom edge, and its middle.

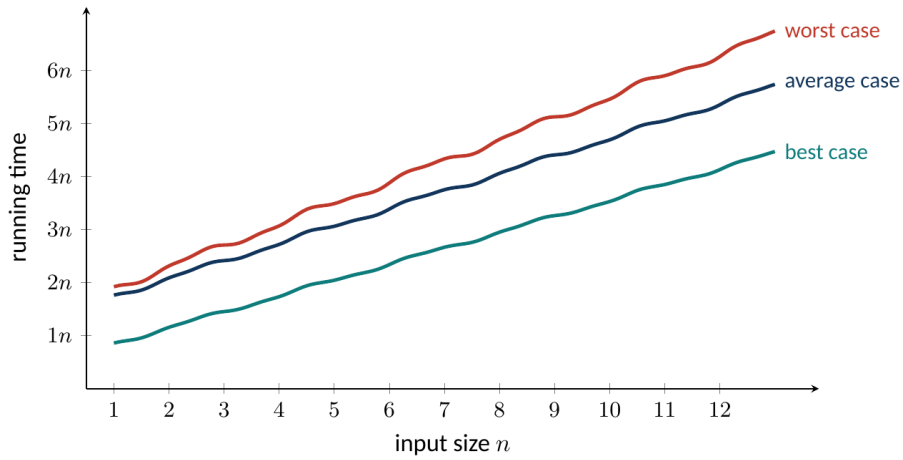


Figure 2.11 For a given input size there are many possible inputs and therefore many possible running times. The worst case is the upper envelope, the best case the lower, and the average case lies between them.

2.11.1 The best case: an array already sorted

Suppose the input array is already in non-decreasing order, for example $A = \langle 1,2,3,4,5,6 \rangle$.

Consider any j. We set $key = A[j]$ and $i = j - 1$, then evaluate the line-4 test. The first condition, $i > 0$, is true since $j \geq 2$. The second condition asks whether $A[j-1] > key = A[j]$. But the array is sorted, so $A[j-1] \leq A[j]$, and the condition is **false**. The test is evaluated once and fails immediately. Hence

$$t_j = 1 \quad \text{for every } j,$$

and therefore $t_j - 1 = 0$: lines 5 and 6 never execute at all.

Substitute into the general formula Equation (2.2). The two sums become

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} 1 = n - 1, \qquad \sum_{j=2}^{n} (t_j - 1) = 0,$$

where the first equality holds because we are adding the number 1 to itself once for each of the $n-1$ values $j = 2,3,\ldots,n$.

So

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 \cdot 0 + c_6 \cdot 0 + c_7(n-1).$$

Collect the terms. Every bracket is $(n-1) = n-1$, so

$$\begin{aligned} T(n) &= c_1 n + (c_2 + c_3 + c_4 + c_7)(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)\, n - (c_2 + c_3 + c_4 + c_7). \end{aligned}$$

Write $a = c_1 + c_2 + c_3 + c_4 + c_7$ and $b = -(c_2 + c_3 + c_4 + c_7)$. Both are constants — sums of constants are constants. Then

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

> ✓ **KEY INSIGHT.** In the best case, Insertion Sort's running time Equation (2.3) is a **linear** function of n. We will write this as $\Theta(n)$ in Chapter 3. Intuitively: the algorithm walks once through the array, confirms at each step that the new element is already in place, and does no shifting at all. It cannot possibly be faster, because it must at least look at every element to know the array is sorted.

2.11.2 The worst case: an array in reverse order

Now suppose the input is in strictly decreasing order, for example $A = \langle 6,5,4,3,2,1 \rangle$.

Consider any j. Every element to the left of position j is larger than $A[j]$ — that is what "decreasing" means. So the line-4 test succeeds for $i = j-1$, for $i = j-2$, and so on all the way down to $i = 1$. It fails only when i reaches 0, on the first condition.

Count the evaluations. The test succeeded for $i = j-1, j-2, \ldots, 1$, which is $j-1$ successes, and then failed once. Therefore

$$t_j = (j-1) + 1 = j.$$

Check it against the definition on a small case. For $j = 2$ the test runs twice: once with $i = 1$ (succeeds, since $A[1] > A[2]$), once with $i = 0$ (fails). And indeed $t_2 = 2$. ✓

Now we must evaluate the sums. First,

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = 2 + 3 + 4 + \cdots + n.$$

This is an **arithmetic series**, and we need its closed form. Rather than quote it, derive it — the trick is worth knowing and is attributed to the young Gauss. Write the sum forward and backward and add the two lines term by term:

$$\begin{array}{rcccccccc} S & = & 2 & + & 3 & + & \cdots & + & n \\ S & = & n & + & (n-1) & + & \cdots & + & 2 \\ \hline 2S & = & (n+2) & + & (n+2) & + & \cdots & + & (n+2) \end{array}$$

Every column adds to $n + 2$, because in the top row the terms increase by one as you move right and in the bottom row they decrease by one, so each column's total is unchanged. How many columns are there? One for each value of j from 2 to n, which is $n - 1$ columns. Hence

$$2S = (n-1)(n+2), \qquad S = \frac{(n-1)(n+2)}{2}.$$

Let us expand it and check it against the more familiar form. Multiplying out, $(n-1)(n+2) = n^2 + n - 2$, so

$$S = \frac{n^2 + n - 2}{2} = \frac{n(n+1)}{2} - 1.$$

That second form is the one to remember, and it has an easy reading: the sum $1 + 2 + \cdots + n$ equals $n(n+1)/2$, and our sum starts at 2 instead of 1, so we subtract the missing 1. Sanity-check with $n = 6$: our sum is $2 + 3 + 4 + 5 + 6 = 20$, and the formula gives $6 \cdot 7/2 - 1 = 21 - 1 = 20$. ✓

So

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

Second sum. Since $t_j - 1 = j - 1$,

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

where the last step is the same Gauss formula applied to $1 + 2 + \cdots + m$ with $m = n - 1$, giving $m(m+1)/2 = (n-1)n/2$. Check with $n = 6$: $1 + 2 + 3 + 4 + 5 = 15$, and $6 \cdot 5/2 = 15$. ✓

Now substitute both, Equation (2.4) and Equation (2.5), into the general formula Equation (2.2):

$$\begin{aligned} &T(n) \\ &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\frac{n(n-1)}{2} \\ &\qquad + c_6\frac{n(n-1)}{2} + c_7(n-1). \end{aligned}$$

Expand the quadratic pieces. Using $\frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$ and $\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$:

$$\begin{aligned} T(n) \\ = c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_4 \frac{n^2}{2} + c_4 \frac{n}{2} - c_4 + c_5 \frac{n^2}{2} - c_5 \frac{n}{2} + c_6 \frac{n^2}{2} \\ - c_6 \frac{n}{2} + c_7 n - c_7. \end{aligned}$$

Collect by power of n. The n^2 terms:

$$\left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n^2.$$

The n terms:

$$\left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right) n.$$

The constant terms:

$$-(c_2 + c_3 + c_4 + c_7).$$

Naming those three brackets a, b and c — all of them constants, because they are built only out of the constants $c_1, \ldots, c_7$ — we have derived

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

In Equation (2.6), $a = (c_4 + c_5 + c_6)/2 > 0$. The worst-case running time of Insertion Sort is a **quadratic** function of n. In Chapter 3's language it is $\Theta(n^2)$, and the derivation above, not anyone's say-so, is why.

✓ **KEY INSIGHT.** Worst case of Insertion Sort: $T(n)$ is quadratic. The practical meaning is brutal. Sorting a reverse-sorted array of one million elements needs about $10^{12}/2$ shift operations. At a very optimistic 10^9 operations per second that is roughly 500 seconds — over eight minutes — to sort a list that `std::sort` handles in well under a second.

⚠ **THINK.** Notice what the derivation did *not* depend on: the values of c_1 through c_7. Whatever your compiler, whatever your processor, the answer has the form $an^2 + bn + c$. The machine changes a; it does not change the n^2. That is why we can do this analysis on paper, years before the program is written, on hardware that does not exist yet.

2.11.3 The average case

Best and worst are the two extremes. What happens on a *typical* array?

To ask that question precisely we must say what "typical" means, because an average is an average over some distribution and different assumptions give different answers. The standard assumption is: **all $n!$ orderings of the input are equally likely.**

Fix a value of j. The element $key = A[j]$ is being inserted into the sorted prefix $A[1 \ldots j-1]$, which holds $j-1$ elements. Under our assumption, `key` is equally likely to belong in any of the j available positions: before all $j-1$ of them, between the first and second, and so on, up to after all

of them. The inner loop's test count t_j is determined by which of those positions it is — landing at the far right gives $t_j = 1$, landing at the far left gives $t_j = j$.

Let us make "on average" exact. If `key` lands k places from the right-hand end of the prefix ($k = 0,1,\dots,j-1$), the loop test succeeds k times and then fails once, so $t_j = k + 1$. Each of the j landing places is equally likely, so t_j is equally likely to be any of $1,2,\dots,j$, and its average is the sum of those values divided by how many there are. By the Gauss formula,

$$\text{average of } t_j = \frac{1}{j} \cdot \frac{j(j+1)}{2} = \frac{j+1}{2} \approx \frac{j}{2}.$$

In words: on average the key travels about halfway along the sorted prefix.

Substituting into the first sum, and using $\sum_{j=2}^{n} 1 = n - 1$:

$$\sum_{j=2}^{n} \frac{j+1}{2} = \frac{1}{2}\left(\sum_{j=2}^{n} j + (n-1)\right) = \frac{1}{2}\left(\frac{n(n+1)}{2} - 1 + n - 1\right) = \frac{n^2 + 3n - 4}{4} \approx \frac{n^2}{4}.$$

(Check with $n = 6$: the average test counts are $\frac{3}{2}, \frac{4}{2}, \frac{5}{2}, \frac{6}{2}, \frac{7}{2}$, which add up to 12.5, and $(36 + 18 - 4)/4 = 12.5$.) In the same way the second sum is, on average, $\sum_{j=2}^{n} \frac{j-1}{2} = \frac{1}{2} \cdot \frac{n(n-1)}{2} \approx \frac{n^2}{4}$. So lines 4, 5 and 6 each run about $n^2/4$ times on a typical input, against about $n^2/2$ times in the worst case: exactly half.

Feeding these counts into Equation (2.2) and collecting terms exactly as in Section 2.11.2, the n^2 coefficient becomes $(c_4 + c_5 + c_6)/4$, which is half of the worst-case a:

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

for the same a as in Equation (2.6), where b' and c' are some other constants. So the average case Equation (2.7) is **also quadratic** — half the worst case, near enough, but with the same n^2.

> ✓ **KEY INSIGHT.** The average case of Insertion Sort has the same *order of growth* as the worst case: both are quadratic. Halving the constant does not help you at scale. If eight minutes is too slow for a million elements, four minutes is also too slow. Only a change in the growth rate — from n^2 to $n\lg n$ — actually solves the problem, and that is what Merge Sort does in Chapter 3.

2.11.4 Checking the analysis against a machine

Everything so far was done on paper. Let us verify it. **Listing 2.3** implements Insertion Sort with a counter incremented at every evaluation of the line-4 test — that is, it measures $\sum_{j=2}^{n} t_j$ directly — and runs it on ascending, descending and randomly shuffled input.

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

```
// 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;
}
```

Its actual output is:

```
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
```

Read that table against the three derivations.

The **best** column is exactly $n-1$ every time: 7,15,31,63,127 for $n = 8,16,32,64,128$. That is Section 2.11.1's $\sum t_j = n - 1$, confirmed to the digit.

The **worst** column is identical to the predicted $n(n+1)/2 - 1$ in every row. Not approximately — identically. Section 2.11.2's derivation is not an estimate; it is an exact count, and the machine agrees.

The **random** column sits near half the worst column: $4453/8255 = 0.539$ at $n = 128$, and $1045/2079 = 0.503$ at $n = 64$. That is Section 2.11.3's factor of one half, within the noise you

expect from a single shuffle. The random column grows the same way the worst column does — quadrupling as n doubles — which is the point: same order, smaller constant.

Now the same claim with a stopwatch instead of a counter. **Listing 2.4** times the worst case for $n = 5{,}000$ up to $n = 80{,}000$, doubling each time, and prints the ratio of each time to the previous one.

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.

```
// 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;
}
```

On the machine used to prepare this book, the program printed:

```
       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
```

The millisecond figures will be different on your computer — that is the whole point of Section 2.6.4, and you should expect them to differ. The **ratio** column is the part that travels. In the run above it reads 3.93,3.98,4.00,4.03; on a busy machine you may see anything from about 3.6 to 4.4, because a stopwatch measures your operating system as well as your program. Repeat the run a few times and the figure it circles is four: every doubling of n multiplies the time by four.

Why four? Because if $T(n) = an^2 + bn + c$ then, for large n, the n^2 term dominates, and

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

A quadratic algorithm quadruples when you double the input. A linear one merely doubles. A logarithmic one adds a constant. Measuring the ratio of Equation (2.8) is the quickest way to find out experimentally what your code's growth rate really is, and it is a technique worth having in your hands: if you believe your code is $O(n\lg n)$ and the ratio comes out at 4, you have a bug in your reasoning or your code.

2.11.5 Why we focus on the worst case

Three cases exist; we spend nearly all our attention on one of them. Here is why.

It is a guarantee. The worst-case running time is an *upper bound*: the algorithm will never, on any input, take longer than this. No input exists that can surprise you. An average tells you what usually happens; a worst case tells you what can never fail to be true, and only the second kind of statement can be relied upon.

Some applications require it. In an air-traffic control system, a cardiac pacemaker, an anti-lock braking controller or a surgical robot, "usually fast enough" is not an engineering standard. You must be able to state a bound that holds every time.

The worst case occurs more often than you would guess. It is comforting to assume inputs are random. They rarely are. A search for an absent record hits linear search's worst case exactly; and real data arrives already sorted, already reverse-sorted, or full of duplicates far more often than chance would suggest. Worse, if an adversary can influence your input — a user submitting a form, a client sending a request — they can *choose* your worst case deliberately. Algorithmic complexity attacks work exactly this way.

The average case is often the same order anyway. We just proved this for Insertion Sort: worst is an^2, average is about $\frac{a}{2}n^2$. Four minutes instead of eight is not a rescue.

Computing the true average is hard, and needs an assumption you usually cannot justify. The average-case analysis of Section 2.11.3 rested on "all $n!$ orderings are equally likely". Is that true of your data? For laundry tickets arriving through the day, absolutely not — they arrive nearly sorted. For readings from a sensor, no. Without the distribution, the average case is undefined; the worst case needs no distribution at all.

> **Intuition.** A delivery company that promises "your package arrives within 5 days" has given you something you can plan around. A company that says "it usually arrives in 3 days" has given you a hope. Worst-case analysis sells guarantees, not hopes.

2.12 The Growth of Functions

We have derived three running times: $an + b$, $\frac{a}{2}n^2 + \cdots$, and $an^2 + bn + c$. Section 2.11.4 showed that the constants a, b, c depend on the machine while the powers of n do not. This section shows why, for large n, *only* the powers of n matter — which is the justification for the asymptotic notation that Chapter 3 will define formally.

2.12.1 Logarithms, in words

One symbol needs defining before we can read the tables. The **logarithm of n to base 2**, written $\log_2 n$ or, in this book, $\lg n$, answers this question: *to what power must I raise 2 to get n?* Equivalently, and more usefully for algorithms: *how many times can I halve n before I reach 1?*

- $\lg 8 = 3$, because $2^3 = 8$; and because $8 \to 4 \to 2 \to 1$ is three halvings.
- $\lg 1024 = 10$, because $2^{10} = 1024$.
- $\lg 1{,}000{,}000 \approx 19.93$, because $2^{20} = 1{,}048{,}576$, a little more than a million.

The crucial property is that the logarithm grows with agonizing slowness. Multiplying n by a thousand adds only about ten to $\lg n$. An algorithm whose cost is $\lg n$ is, for practical purposes, free.

2.12.2 The comparison table

Table 2.3 compares five growth rates at four input sizes. Every entry in it was computed by the program in **Listing 2.5** below, so you can check any of them yourself.

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}

Now read the last row, $n = 1{,}000{,}000$, across, and let it sink in. Assume a machine doing 10^9 simple operations per second — a fair figure for one core of a modern processor running compiled C++.

- An $O(\lg n)$ algorithm does **20** operations. The answer arrives before you lift your finger from the key.
- An $O(n)$ algorithm does **one million** operations: about one millisecond.
- An $O(n\lg n)$ algorithm does **twenty million** operations: about twenty milliseconds. Still instant to a human.
- An $O(n^2)$ algorithm does **one trillion** (10^{12}) operations: about 1,000 seconds, which is sixteen minutes and forty seconds. Your user has closed the tab.
- An $O(2^n)$ algorithm does about 10^{301030} operations. There is no unit of time for this. Scientists estimate the observable universe contains between 10^{78} and 10^{82} atoms — a figure often quoted as 10^{80} and known as the Eddington number. The operation count here exceeds the number of atoms in the universe raised to the power of three thousand seven hundred. Multiplying your processor speed by a billion would remove nine digits from an exponent of 301,029. Exponential algorithms are not slow; they are impossible.

Here is the program that produced those numbers. Note the trick in `powerOfTwo`: $2^{1000000}$ cannot be stored in any floating-point type, so we never form it, and instead compute $\log_{10}(2^n) = n\log_{10} 2$ and split it into a mantissa and an exponent.

Listing 2.5 Computing the growth-rate comparison, including 2^{10^6}, without ever storing a number that large.

```
// The growth-rate table, computed rather than quoted.
// log2 of a big number is computed in logarithms so that 2^n never has to be
// stored: log10(2^n) = n * log10(2), and we print mantissa and exponent.
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

void powerOfTwo(double n) {                  // print 2^n as m x 10^e
    double log10val = n * log10(2.0);
    double e = floor(log10val);
    double m = pow(10.0, log10val - e);      // the leading digits
    if (e < 6) cout << setw(14) << (long long)llround(pow(2.0, n));
    else       cout << setw(8) << fixed << setprecision(2) << m
                    << " x 10^" << (long long)e;
}

int main() {
    cout << setw(10) << "n" << setw(10) << "log2 n" << setw(14) << "n"
         << setw(14) << "n log2 n" << setw(16) << "n^2"
         << "   2^n\n";
    for (double n : {10.0, 100.0, 1000.0, 1e6}) {
        double lg = log2(n);
        cout << setw(10) << (long long)n
             << setw(10) << fixed << setprecision(1) << lg
             << setw(14) << (long long)n
             << setw(14) << (long long)llround(n * lg)
             << setw(16) << (long long)(n * n) << "   ";
        powerOfTwo(n);
        cout << '\n';
    }
    return 0;
}
```

Its actual output is:

```
         n    log2 n             n      n log2 n             n^2   2^n
        10       3.3            10            33             100             1024
       100       6.6           100           664           10000       1.27 x 10^30
      1000      10.0          1000          9966         1000000       1.07 x 10^301
   1000000      19.9       1000000      19931569   1000000000000       9.90 x 10^301029
```

> ✓ **KEY INSIGHT.** As n grows, the **rate of growth** dominates everything. Constants and lower-order terms become irrelevant. An n^2 algorithm with a wonderful constant of 0.001 still loses to an $n\lg n$ algorithm with a terrible constant of 1000 — it just takes a larger n to show it. This is exactly why asymptotic notation exists: it throws away the part of the answer that depends on your hardware and keeps the part that decides whether your program finishes.

2.12.3 Seeing it as a picture

Two graphs make the point that a table cannot. **Figure 2.12** plots the common polynomial and logarithmic running times for n from 2 to 1024, with *both* axes logarithmic: along the bottom each tick doubles n, and up the side each gridline multiplies $T(n)$ by ten. (The curve marked $\lg n$ is the base-2 logarithm, so it reads 10 at $n = 1024$; a logarithm to any other base would only be a constant multiple of it, with the same shape.)

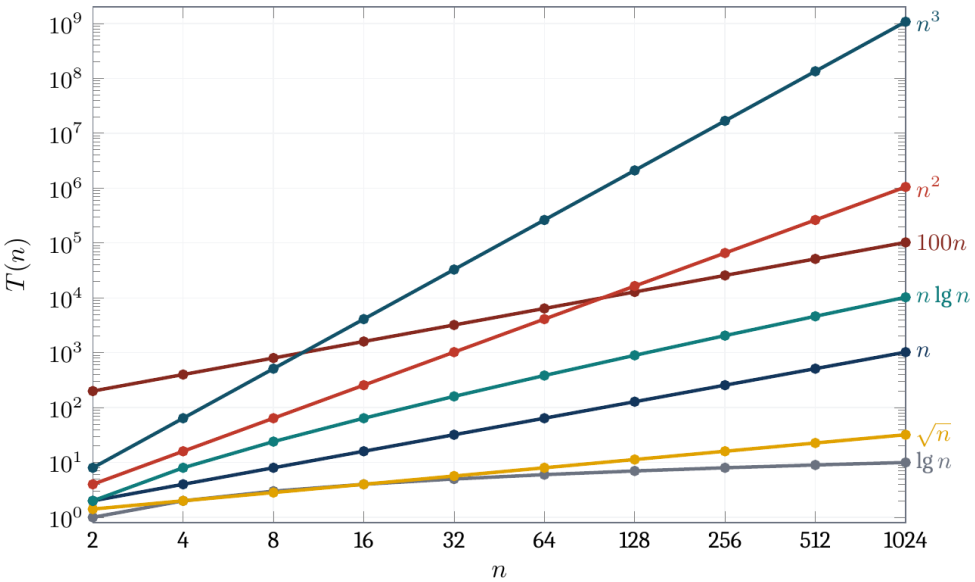


Figure 2.12 Growth of $\lg n$, $\sqrt{n}$, n, $n\lg n$, $100n$, n^2 and n^3 with both axes logarithmic. On such a plot a function n^k becomes a straight line whose slope is k, so steeper means worse.

On a log–log plot every function of the form n^k becomes a straight line of slope k, which makes the picture easy to read: n^3 is the steepest line, then n^2, then n, and $\lg n$ is nearly flat. Look at the line marked $100n$. At $n = 2$ it starts at 200, far above n^2, which starts at 4. A student comparing the two algorithms on tiny inputs would conclude that the quadratic one is fifty times better. But the lines cross near $n = 100$, and after that n^2 climbs away forever. **A large constant can hide a bad growth rate, but only for small n.** This is also why an $O(n^2)$ algorithm is often genuinely the right choice when you know n will always be small.

Now add the exponential, and the whole picture changes. **Figure 2.13** is the same plot with 2^n included, and the vertical axis must stretch to about 10^{308} to fit it, because $2^{1024} \approx 1.8 \times 10^{308}$. The vertical axis is again logarithmic, labeled in powers of ten.

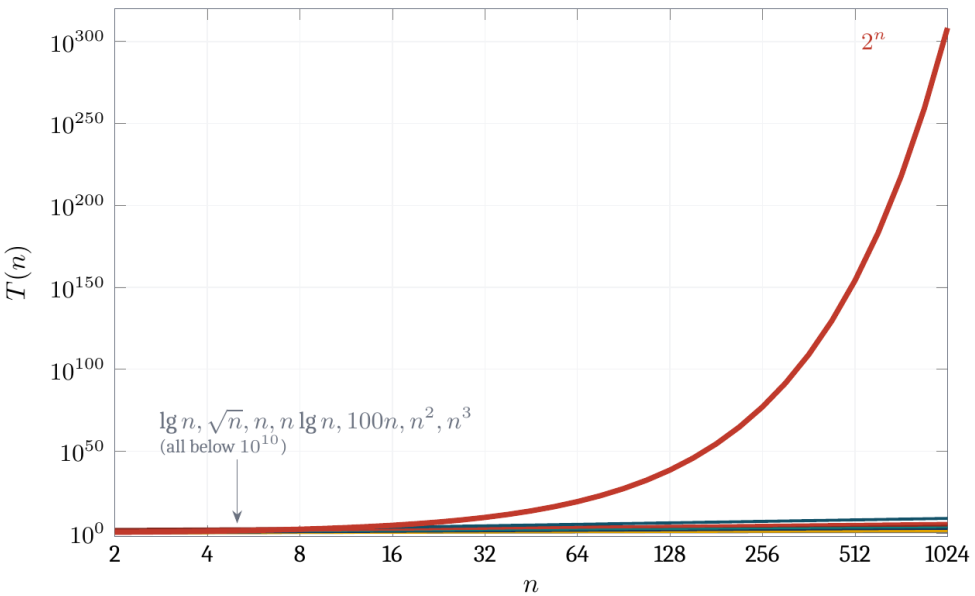


Figure 2.13 The same growth rates with 2^n added. The vertical axis now runs to about 10^{308}; every polynomial curve is crushed into the bottom of the chart, indistinguishable from the axis.

Everything we were carefully comparing in **Figure 2.12** — the difference between n, $n\lg n$, n^2 and n^3 — has been flattened into a single indistinguishable smear along the bottom of **Figure 2.13**. Against an exponential, all polynomials are the same thing: fast. That is the deepest lesson of the two pictures, and it is why theoretical computer science draws its main dividing line between polynomial and exponential rather than between n and n^2.

2.12.4 The hierarchy of growth rates

From slowest-growing to fastest-growing — that is, from best to worst as a running time:

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

Here the sign $<$ between two classes means "grows strictly more slowly than, once n is large enough". Equation (2.9) is worth knowing by heart, with a plain example of each class:

- $O(1)$, **constant.** Does not depend on n at all. Reading `A[i]`; pushing onto a stack; arithmetic.
- $O(\lg n)$, **logarithmic.** Binary search (Section 2.14); inserting into a balanced tree (Chapter 6).
- $O(\sqrt{n})$, **root.** Trial division to test whether n is prime.
- $O(n)$, **linear.** One pass over an array: linear search, finding the maximum, summing.
- $O(n\lg n)$, **linearithmic.** The sweet spot for sorting: Merge Sort, Heapsort, Quicksort on average.
- $O(n^2)$, **quadratic.** Insertion Sort's worst case; comparing every pair of n items.
- $O(n^3)$, **cubic.** Naive matrix multiplication; Floyd–Warshall all-pairs shortest paths (Chapter 15).
- $O(2^n)$, **exponential.** Enumerating every subset of n items. Hopeless beyond $n \approx 30$.
- $O(n!)$, **factorial.** Enumerating every ordering of n items — the "sort by trying all permutations" algorithm of Section 2.4.3. Hopeless beyond $n \approx 12$.

2.12.5 A practical rule of thumb

For competitive programming and for engineering judgment alike, the useful form of all this is a lookup table. A modern machine performs roughly 10^8 to 10^9 simple operations per second. Given a typical time limit of one or two seconds, **Table 2.4** says which complexities are affordable at which input sizes.

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 it in the direction that matters. The problem statement tells you the maximum n before you have written a line of code. Look n up in **Table 2.4**, read off the complexity you are allowed, and only then start designing. If the constraint says $n \le 200{,}000$ and your idea is $O(n^2)$, you already know it will fail — you have saved yourself an hour of implementation and a wrong answer.

⚠ **THINK.** If your algorithm is $O(n^2)$ and $n = 1{,}000{,}000$, it needs 10^{12} operations — around 1,000 seconds, or sixteen minutes and forty seconds, against a two-second limit. No amount of micro-optimization closes a gap of five hundred times. You do not need faster code; you need a different algorithm.

2.13 Searching: Linear Search

Sorting arranges data. Searching finds things in it. Searching is the second fundamental problem of the chapter, and it will show us something sorting did not: that the *structure* of the data can change what is achievable, not merely by a constant factor but by an entire order of growth.

2.13.1 The problem

THE SEARCHING PROBLEM.

Input: a sequence $A = \langle a_1, a_2, \dots, a_n \rangle$ and a query value q.

Output: an index j such that $A[j] = q$, or the special value NIL if q does not occur in A.

Figure 2.14 states this in input/output form, with two concrete instances underneath.

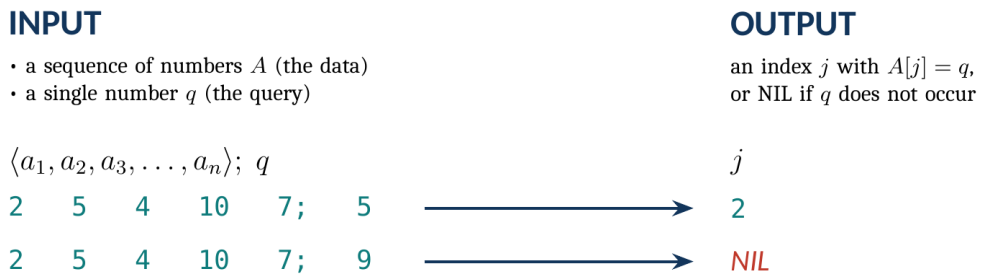


Figure 2.14 The searching problem. On the instance $A = \langle 2,5,4,10,7 \rangle$ with $q = 5$ the answer is the index 2; with $q = 9$ the answer is NIL, because 9 does not occur.

Two details of that specification are worth pausing on.

First, the output is an **index**, not the value. Returning the value would be pointless — the caller already has q; what they need is *where* it is, so they can look at the record stored alongside it.

Second, NIL is not a number. It is a marker meaning "no such index". In C++ we must encode it as some value the caller can distinguish from a real answer. With 1-based indexing we use 0, because 0 is never a legal index in our scheme; with 0-based indexing the usual convention is -1. What you must never do is return something like 0 in a 0-based program, where the caller cannot tell "found at the front" from "not found at all".

2.13.2 The algorithm

If the array is not sorted, we have no choice: we must look at the elements one at a time until we find q or run out of array. **Pseudocode 2.3** does exactly that.

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
```

Once again the order of the two conditions on line 2 is load-bearing: `j <= A.length` must be tested **first**, so that when j runs past the end we stop before evaluating $A[j]$ on a position that does not exist. In C++, `&&` guarantees this left-to-right short-circuit evaluation.

Line 4 then asks why the loop stopped. If it stopped because the second condition failed, then j is a legal index and $A[j] = q$: we found it. If it stopped because the first condition failed, then $j = n + 1$, which is past the end, and q is absent.

2.13.3 A loop invariant for linear search

The same proof technique applies, and it is short enough to do in a paragraph.

> **LOOP INVARIANT (linear search).** At the start of each iteration of the `while` loop, none of $A[1], A[2], \ldots, A[j-1]$ is equal to q.

Initialization. Before the first iteration $j = 1$, so the claim concerns $A[1 \ldots 0]$, an empty range. No element of an empty range equals q, vacuously. ✓

Maintenance. Suppose the invariant holds and the loop body runs. The body runs only if the test succeeded, which required $A[j] \neq q$. So now none of $A[1 \ldots j-1]$ equals q *and* $A[j] \neq q$, which together say none of $A[1 \ldots j]$ equals q. Line 3 increments j, so the invariant with the new j reads "none of $A[1 \ldots j-1]$ equals q" — exactly what we just established. ✓

Termination. Two exits, matching lines 4–7. If the loop stopped because $A[j] = q$, we return j and $A[j] = q$ is the required output. If it stopped because $j > n$, then $j = n + 1$ and the invariant says none of $A[1 \ldots n]$ equals q — the whole array — so NIL is correct. ✓ ■

2.13.4 Analysis

Line 2's test is the operation that matters; everything else is a constant multiple of it.

Worst case. If q does not occur in A, the loop runs until $j = n + 1$, evaluating the test $n + 1$ times and the body n times. So $T(n) = \Theta(n)$. The same happens if q is at the very last position.

Best case. If $A[1] = q$, the test is evaluated once and we return immediately: $T(n) = \Theta(1)$.

Average case. Assume q is present and equally likely to be at each of the n positions. Then the number of comparisons is 1 with probability $1/n$, 2 with probability $1/n$, and so on, so the expected number is

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

using the Gauss formula we derived in Section 2.11.2. About half the array — and the value $\frac{n+1}{2}$ in Equation (2.10) is still $\Theta(n)$.

> ✓ **KEY INSIGHT.** Linear search on an **unsorted** array is $\Theta(n)$ in the worst case, and this is optimal. The argument is short and worth internalizing: if the algorithm has not examined

position k, an adversary is free to place q there, so any correct algorithm must examine every position before it may answer NIL. You cannot beat $\Theta(n)$ on unsorted data. To go faster, you must change the data, not the algorithm.

2.13.5 Linear search in C++

Listing 2.6 Linear search in C++, on the same data as **Figure 2.14**.

```
// 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;
}
```

The output of **Listing 2.6** is:

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

which is exactly the two answers printed in the OUTPUT column of **Figure 2.14**.

2.13.6 Can we do better?

? QUESTION. We have just proved that $\Theta(n)$ is the best possible on an unsorted array. But what if the array is **sorted**? Does the extra structure buy us anything?

It buys us an enormous amount. This is the first time in the book we see the central trade of the subject: *invest in structure, and get speed back.* The reason is easy to state. In an unsorted array, looking at $A[5]$ and finding it is not q tells you exactly one thing: q is not at position 5. In a sorted array, looking at $A[5]$ and finding that $A[5] > q$ tells you that q is not at position 5 — and also not at 6, 7, 8, or anywhere further right, because everything there is even larger. **One comparison eliminates many positions.** That is the whole idea of the next section.

2.14 Binary Search

2.14.1 The problem

THE SORTED SEARCHING PROBLEM.

Input: a **sorted**, non-decreasing sequence $A = \langle a_1 \le a_2 \le \cdots \le a_n \rangle$ and a query value q.

> *Output:* an index j such that $A[j] = q$, or NIL if q does not occur in A.

The only change from Section 2.13.1 is the word *sorted* in the input line, as **Figure 2.15** shows. Look at the two instances at the bottom of that figure and compare them with the ones in **Figure 2.14**: the same five numbers, but now in order, so that $q = 5$ is answered with index 3 rather than index 2. The answer to a search depends on the arrangement; the *problem* is a different problem.

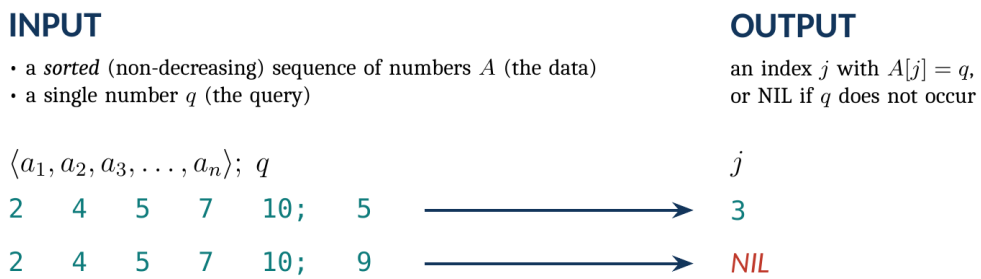


Figure 2.15 The sorted searching problem. The one added word — *sorted* — in the input specification is what makes an exponential speed-up possible.

> ⚠ **THINK.** Binary search on an unsorted array does not merely run slowly; it returns **wrong answers**, silently. It will confidently report NIL for a value that is sitting in the array. The precondition is not advice, it is a requirement, and nothing in the code checks it for you.

2.14.2 The key idea

Compare q with the **middle** element of the array. Exactly one of three things happens.

- If $A[mid] = q$, we have found it. Return mid.
- If $A[mid] > q$, then q must lie in the left half, $A[1 \ldots mid - 1]$. Why? Because the array is sorted, so every element from mid rightwards is $\geq A[mid] > q$, and none of them can equal q. Discard the entire right half.
- If $A[mid] < q$, then by the mirror-image argument q can only lie in $A[mid + 1 \ldots n]$. Discard the entire left half.

Each comparison **halves** the search space. This is our first real example of the divide-and-conquer paradigm, and Chapter 3 will build the whole theory of recurrences on this shape.

There is a familiar human version. To look up a surname beginning with S in a printed phone book you do not start at page 1. You open the book near the middle, see that you have landed in the M's, and throw away the first half without reading a single name in it. That discarding is the algorithm.

2.14.3 The algorithm

Pseudocode 2.4 turns the key idea into a procedure.

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
```

The pair $(left, right)$ defines a **window**: the portion of the array in which q could still be hiding. Initially the window is the whole array. Each iteration shrinks it. The loop ends when the window becomes empty, which happens when $left > right$, and an empty window means q is not there.

The invariant is the natural one, and you should now be able to guess it before reading it.

> **LOOP INVARIANT (binary search).** At the start of each iteration, if q occurs anywhere in A, then it occurs at some index within the window $[\,left, right\,]$.

Initialization. Before the first iteration $left = 1$ and $right = n$, so the window is the whole array. If q occurs in A it certainly occurs in the whole array. ✓

Maintenance. Assume the invariant and suppose we do not return on line 5. If $A[mid] > q$, then since A is sorted, $A[k] \geq A[mid] > q$ for every $k \geq mid$, so q occurs at no index $\geq mid$. Combined with the invariant, if q occurs at all it occurs in $[\,left, mid - 1\,]$ — which is exactly the new window set by line 6. The case $A[mid] < q$ is symmetric. ✓

Termination. The loop exits when $left > right$, that is, when the window is empty. The invariant then says: if q occurs in A, it occurs in an empty range. That is impossible, so q does not occur in A, and line 8's NIL is correct. The other exit is line 5, which returns mid having just tested $A[mid] = q$. ✓

Does the loop terminate at all? Yes, and it is worth checking rather than assuming. The window size is $right - left + 1$. Each iteration sets either $right \leftarrow mid - 1$ or $left \leftarrow mid + 1$, and since $left \leq mid \leq right$, both assignments strictly shrink the window by at least one. A quantity that is a non-negative integer and strictly decreases every iteration must reach zero in finite time. ✓ ■

> ⚠ **TRAP.** The `+1` and `-1` on lines 6 and 7 are not decoration. They exclude mid itself, which we have just tested and ruled out. Write `right = mid` instead of `right = mid - 1` and consider a two-element window with $left = 1$, $right = 2$: then $mid = 1$, and if $A[1] > q$ we set $right = 1$, giving the window $[1,1]$ again — the same window, forever. The program hangs. The `-1` is what guarantees progress, and therefore what makes the termination argument above valid.

2.14.4 Tracing binary search by hand

Search for $q = 7$ in $A = \langle 2,4,5,7,10 \rangle$, which has $n = 5$. **Table 2.5** records the trace.

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

Expanding that, one step per line:

1. $left \leftarrow 1, right \leftarrow 5$. Window is $[1,5]$, holding $\langle 2,4,5,7,10 \rangle$.

2. Test $left \leq right$: $1 \leq 5$, true. Enter the loop.
3. $mid \leftarrow (1+5)/2 = 6/2 = 3$.
4. Is $A[3] = 7$? $A[3] = 5$, no.
5. Is $A[3] > 7$? $5 > 7$ is false, no.
6. So $A[3] < 7$: set $left \leftarrow mid + 1 = 4$. The window is now [4,5]. We have discarded positions 1, 2 and 3 — three of the five — with a single comparison.
7. Test $left \leq right$: $4 \leq 5$, true. Enter the loop again.
8. $mid \leftarrow (4+5)/2 = 9/2 = 4$ by integer division, which truncates toward zero.
9. Is $A[4] = 7$? $A[4] = 7$, yes. Return 4. ✓

Two comparisons rather than the four that linear search would have needed to reach position 4. On five elements that is unimpressive. On a million it is the difference between 20 and 1,000,000.

2.14.5 Why it is $O(\lg n)$

> ❓ **QUESTION.** How many times can you cut n in half before there is nothing left to cut?

Let us count. Start with a window of n elements.

- After 1 comparison, the window holds at most $n/2$ elements.
- After 2 comparisons, at most $n/4$.
- After 3 comparisons, at most $n/8$.
- After k comparisons, at most $n/2^k$.

(In fact the window shrinks slightly faster than this, because mid itself is discarded too, but $n/2^k$ is a valid upper bound and the extra element does not change the conclusion.)

The loop must stop once the window can hold no elements, and certainly once $n/2^k \leq 1$. Solve that for k:

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

The middle step of Equation (2.11) multiplies both sides by 2^k, which is positive so the inequality direction is unchanged; the last step takes $\log_2$ of both sides, which is legitimate because $\log_2$ is an increasing function.

So $k \geq \lg n$ comparisons suffice, and the worst-case running time is

$$T(n) = \Theta(\lg n).$$

Put numbers on it. For $n = 1{,}000{,}000$, $\lg n \approx 19.93$, so at most **20 comparisons** — against up to a million for linear search. For $n = 10^{18}$, a number larger than the count of grains of sand on Earth, $\lg n \approx 60$. Sixty comparisons. This is the row at the bottom of **Table 2.4**.

> ✓ **KEY INSIGHT.** Binary search runs in $\Theta(\lg n)$ time. Every doubling of the input adds exactly **one** comparison. This is what "exploiting structure" buys: not a better constant, but a different function. And it came from one word — *sorted* — in the input specification.

2.14.6 Binary search in C++, and the overflow trap

Listing 2.7 Binary search in C++, iterative form, instrumented to print the window at each step.

```
// 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;
}
```

Its actual output is:

```
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 two traced steps are identical to **Table 2.5**, and the last two lines reproduce the OUTPUT column of **Figure 2.15**: on the sorted data, $q = 5$ is at index 3 and $q = 9$ is absent.

Now look again at line 4 of **Pseudocode 2.4**. It says $mid = (left + right)/2$, and the program does **not** say that. It says `left + (right - left) / 2`. The two expressions are equal in ordinary arithmetic. They are not equal in `int` arithmetic, and the difference is a famous bug: it lay undetected for about two decades in the binary search of *Programming Pearls* [2], and for nearly a decade in the Java standard library's binary search, which had copied it.

The reason is that $left + right$ can exceed the largest representable `int` even though $left$, $right$ and the true midpoint all fit comfortably. **Listing 2.8** demonstrates it rather than asserting it.

Listing 2.8 Why the midpoint must be computed as `left + (right - left) / 2`.

```
// 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;
}
```

Its actual output is:

```
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
```

The naive formula produces $-397{,}483{,}648$: a negative index, which as an array subscript is a crash at best and silent corruption at worst. The safe formula works because $right - left$ is at most the size of the array and therefore always small, so nothing ever exceeds the range.

> ⚠ **TRAP.** Always write `mid = left + (right - left) / 2`. It costs nothing, it reads almost as clearly, and it is correct for every input. Make it a habit now, while you are learning, and you will never have to remember why.

2.14.7 Binary search as recursion

Binary search was our first example of divide and conquer, and divide and conquer is naturally recursive. Writing it that way makes the structure visible, and it is the form Chapter 3 will analyze with a recurrence.

Listing 2.9 Binary search as a recursive divide-and-conquer procedure, printing the window at each level of recursion.

```
// 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;
}
```

The output of **Listing 2.9** is:

```
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
```

Read the indentation: each level of the recursion is one comparison, and the window shrinks $8 \to 4 \to 2 \to 1 \to 0$. With $n = 8$ we have $\lg 8 = 3$, and indeed the successful search took 3 comparisons and the unsuccessful one took 4 (three comparisons plus the empty-window check). That matches the $\lg n$ prediction of Section 2.14.5 exactly.

Notice also the shape of the recursion: **one** recursive call, on a problem of half the size, plus constant work. That is the recurrence

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

whose solution is $\Theta(\lg n)$. Recurrence Equation (2.12) says: the time for a window of n elements is the time for a window of half the size, plus a constant for the one comparison. You will solve recurrences like this mechanically with the Master Theorem in Chapter 3; for now it is enough to see where the recurrence comes from.

2.14.8 The trade-off: preprocessing

Binary search requires a sorted array. Someone has to do the sorting, and sorting costs $\Theta(n\lg n)$ with a good algorithm. So is binary search worth it?

It depends entirely on how many searches you do. Let k be the number of searches.

- **Linear search:** no preprocessing, $\Theta(n)$ per search. Total: $\Theta(kn)$.
- **Sort, then binary search:** $\Theta(n\lg n)$ once, then $\Theta(\lg n)$ per search. Total: $\Theta(n\lg n + k\lg n)$.

Compare them for $n = 1{,}000{,}000$, counting in units of 10^6 operations:

- $k = 1$ **search.** Linear: 1 unit. Sort-and-search: about 20 units. **Linear wins** — never sort an array to search it once.
- $k = 100$ **searches.** Linear: 100 units. Sort-and-search: $20 + 100 \times 0.00002 \approx 20$ units. **Binary wins**, five times over.
- $k = 10{,}000$ **searches.** Linear: 10,000 units. Sort-and-search: still about 20 units. **Binary wins overwhelmingly**, by a factor of five hundred.

Setting the two totals equal and solving for k shows the break-even point is at roughly $k \approx \lg n$ searches — about 20 for a million elements. Beyond that, sorting pays for itself.

> ✓ **KEY INSIGHT.** This is the first appearance of a pattern that recurs throughout the book: **pay once to build structure, then query cheaply forever.** Hash tables, balanced search trees, heaps, prefix-sum arrays, suffix arrays and graph preprocessing all make the same bargain. The question is never "is the structure worth building?" in the abstract; it is "how many queries will amortize it?"

2.15 Design Paradigms Revisited

Chapter 1 named the paradigms. We have now seen two of them work, in full detail, and can say what distinguishes them.

2.15.1 The incremental approach

Build the solution one piece at a time: having solved the problem for the first $j - 1$ items, do a little work to extend the solution to j items. Repeat until $j = n$.

Insertion Sort is the canonical example. Its loop invariant *is* the statement of the incremental method: "$A[1 \ldots j - 1]$ is already sorted", and each iteration extends the sorted part by one.

Strengths. Simple to understand, simple to implement, simple to get right. Excellent on small inputs, where its low constant beats fancier algorithms — this is why real library sorts such as `std::sort` switch to Insertion Sort for subarrays of about sixteen elements or fewer. Excellent on nearly-sorted data, where it approaches its $\Theta(n)$ best case. It is also **online**: it can sort data as it arrives, without knowing in advance how much there will be.

Weakness. It often leads to $\Theta(n^2)$ algorithms, because extending the solution by one element may require touching all the elements already processed: $1 + 2 + \cdots + n$ again.

2.15.2 Divide and conquer

Break the problem into smaller sub-problems **of the same kind**, solve those recursively, and combine the answers. Three steps, always: *divide*, *conquer*, *combine*.

Binary Search is the simplest possible example. Divide: compute mid and decide which half can contain q. Conquer: search that half recursively. Combine: nothing to do — the sub-problem's answer is the answer. Because it throws one half away instead of solving both, it is sometimes called *decrease and conquer*.

Merge Sort, which you meet at the start of Chapter 3, is the full version. Divide: split the array into two halves. Conquer: sort each half recursively. Combine: merge the two sorted halves in linear time. Running time $\Theta(n\lg n)$ — dramatically better than Insertion Sort's $\Theta(n^2)$ for large n.

Strengths. Often yields $\Theta(n\lg n)$ or $\Theta(\lg n)$ where the incremental approach yields $\Theta(n^2)$. Naturally recursive, so the code is often shorter than the iterative equivalent. Parallelizes well, since independent sub-problems can be solved simultaneously.

Weaknesses. Recursion has overhead — a function call is not free — and for small inputs that overhead can exceed the savings. It often needs extra memory for the combine step: Merge Sort needs $\Theta(n)$ scratch space, while Insertion Sort needs none.

2.15.3 Ad hoc algorithms

The Latin phrase *ad hoc* means "for this" — a solution designed for one specific problem, exploiting a mathematical structure peculiar to it, which does not generalize to anything else.

Chapter 1 contained two:

- **Columns:** the answer is $\max(n, \max_i a_i)$. A direct formula, arrived at by noticing a property of the problem, not by applying a paradigm.
- **Interesting Sequence:** the answer is $\text{popcount}(m-1) \bmod 3$. A bit-manipulation identity specific to how that particular sequence is constructed.

Ad hoc solutions are usually the fastest code you will ever write — often $O(1)$ — and they are the hardest to find, because there is no method. You find them by staring at small cases until a pattern appears, which is exactly what Phase 2 of the Five-Phase Framework asks you to do.

> ✓ **KEY INSIGHT.** Paradigms are what you reach for when you do not yet see the trick. They are general, teachable and reliable. Ad hoc insights are faster when they exist, but you cannot plan to have one. Learn the paradigms so you always have a move; keep looking for the trick so that sometimes you have a better one.

2.16 Connecting Chapter 1 and Chapter 2

In Chapter 1 you solved six problems and met the Five-Phase Framework, but the complexity claims we made about those solutions were assertions. You can now prove every one of them. **Table 2.6** revisits all six with the machinery of this chapter applied.

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	$O(\lg^2 n \cdot \lg\lg n)$

Three of those rows deserve a comment.

Columns and Journey are $O(n)$ and cannot be improved, for the reason we proved about linear search in Section 2.13.4: any correct algorithm must at least read its input, and the input has n numbers in it. Reading n numbers costs n steps. A "lower bound" argument of this kind — no algorithm whatsoever can do better — is one of the most useful things you can establish, because it tells you when to stop optimizing.

Interesting Sequence is $O(1)$ because `__builtin_popcount` compiles to a single machine instruction on any modern processor. This is the ad hoc paradigm of Section 2.15.3 at its most extreme: the answer does not depend on n at all.

The Recursive Function 1 row shows why input size must be defined carefully (Section 2.7.1). Every argument the recursion ever meets has the form $\lfloor n/(2^a 3^b) \rfloor$ with $a \leq \lg n$ and $b \leq \log_3 n$, so the memo table holds $D = O(\lg^2 n)$ distinct keys — 1,012 of them for $n = 10^{18}$, as Section 1.11.2 measured. Each key is computed once, and each `std::map` lookup costs $O(\lg D) = O(\lg\lg n)$. Multiply them out to get the entry in the table. Without memoization the same recursion makes $2f(n) - 1$ calls, about 4.5×10^{14} for $n = 10^{18}$ — the difference between instant and fifty-two days.

2.16.1 Where this chapter sits in the book

Table 2.7 maps the topics of this book onto CLRS [1], so you always know where to read further.

Table 2.7 The topics of this book mapped onto *Introduction to Algorithms*, 4th edition (computational geometry appears only in the 3rd edition). This chapter is rows one and two.

Topic	CLRS chapters	Key concepts
Introduction and getting started	1–2	Insertion Sort, loop invariants
Correctness and asymptotic analysis	2–4	Big-O, Θ, Ω, solving recurrences
Heapsort and Quicksort	6–7	Heap data structure, partitioning
AVL trees	13	Balanced BSTs, rotations
Dynamic programming	14	Optimal substructure, memoization
Greedy algorithms	15	Activity selection, Huffman coding
Graphs	20	BFS, DFS, topological sort
MST and shortest paths	21–23	Prim, Kruskal, Dijkstra, Bellman–Ford, Floyd–Warshall
Amortized analysis	16	Aggregate, accounting, potential method
Computational geometry (further reading)	33 of the 3rd edition	Convex hull, line intersection

In Chapter 3 we make precise everything this chapter did informally: the formal definitions of O, Θ and Ω; how to *prove* that $f(n) = O(g(n))$ rather than assert it; how to solve recurrences by substitution, recursion trees and the Master Theorem; and the analysis of Merge Sort, $T(n) = 2T(n/2) + \Theta(n) \Rightarrow \Theta(n\lg n)$, which finally beats the quadratic we derived here.

2.17 Profitina in Practice

Profitina is an AI-visibility platform: it scans a website, scores how ready each page is to be **cited** by an AI answer engine, and explains every signal behind the score. The product is live at app.profitina.com, built in Python over PostgreSQL, with a local Qwen2.5 model served through Ollama for the content engine. None of that sounds like Chapter 2 material. It is, in four places.

Input size is what your constraint is written in. A brand's site may have 40 pages or 40,000. Every measurement in the scanner has to be described as a function of the page count n, the average page length m, or both — Section 2.7.1's lesson, straight out of the textbook. A signal check that is $\Theta(n)$ runs on every scan without anyone noticing. A check that compares each page against every other page is $\Theta(n^2)$: at 40 pages that is 1,600 comparisons and invisible; at 40,000 it is 1.6 billion, and by **Table 2.4** that is minutes of CPU per scan. The scanner therefore has a stated complexity budget per signal, and a signal that cannot be computed within it does not ship. This is **Table 2.4** used exactly as Section 2.12.5 describes: the constraint is known first, and it decides the algorithm.

Nearly-sorted data makes the incremental method the right one. The historical analytics dashboard keeps a weekly score trend per brand. Scores arrive in time order, almost always — but not always, because a re-scan that failed and was retried, or a backfill after an outage, can deliver a week out of turn. Maintaining the series by inserting each new point into its place is the incremental method of Section 2.15.1, and on nearly-sorted input it runs at close to its $\Theta(n)$ best case (Section 2.11.1), not its $\Theta(n^2)$ worst. Choosing a "better" $\Theta(n\lg n)$ sort here would be slower in practice: this is the case **Figure 2.12** illustrates, where the constant decides.

Binary search is what makes the scheduler affordable. The re-scan scheduler must repeatedly ask "which brands are due now?" against a list kept ordered by next-due time. Sorting once and

searching many times is the bargain of Section 2.14.8, with $k \gg \lg n$ — the scheduler queries far more often than the list changes, so the preprocessing pays for itself many times over each day.

Worst case is not pessimism; it is the SLA. A scan has a timeout. A weekly trend query has a page-load budget. Section 2.11.5's argument is the operational one: the average scan time tells you what usually happens, and the worst-case bound tells you what you can promise. And because a scan runs against a site *someone else controls*, the input is adversarial by construction — a page can be arbitrarily long, deeply nested, or full of near-duplicate sections. That is precisely the situation in which an average-case argument is worthless.

> **A word on responsibility.** Anything Profitina computes about a page is a measurement and an explanation, never a guarantee about how an AI engine will behave. The app advises; the user decides. And any scan touching personal data is bound by UU PDP 27/2022 — consent must be explicit and understood, not merely clicked.

2.17.1 The runnable case study: Profitina Laundry

The hands-on case study in this book is always **Profitina Laundry**, a small teaching business, never the real product. Here is this chapter's material at work in it.

Profitina Laundry takes in bundles of washing all day. Each customer gets a numbered ticket. The counter book must be kept ordered by ticket number, because when a customer returns to collect, the staff must find their ticket fast. Tickets are issued in order, so arrivals are *nearly* sorted — but occasionally a customer changes service and is re-booked, which puts one ticket out of sequence.

That is exactly the situation the incremental method was made for. **Listing 2.10** keeps the book sorted by inserting each arrival into place, and finds tickets by binary search. It counts both costs so the two can be compared.

Listing 2.10 Profitina Laundry's counter book: incremental insertion to keep it ordered, binary search to find a ticket.

```
// Profitina Laundry: the day's counter book.
// Orders arrive through the day, each with a ticket number. The book is kept
// sorted by ticket number at all times using the incremental method
// (Insertion Sort's inner loop), and a customer collecting an order is found
// with binary search. We count the work so the two costs can be compared.
#include <iostream>
#include <vector>
#include <string>
#include <iomanip>
using namespace std;

struct Order {
    int    ticket;       // the number printed on the customer's slip
    double kg;           // weight of the bundle
    string service;      // "reguler", "kilat", "setrika"
};

long long shifts = 0;    // total data movements done by all insertions

// Insert one order into an already-sorted book. This is exactly the body of
// the Insertion Sort outer loop, used one arrival at a time.
void insertOrder(vector<Order>& book, const Order& o) {
    book.push_back(o);                       // make room at the end
    int i = (int)book.size() - 2;            // last element of the sorted part
    while (i >= 0 && book[i].ticket > o.ticket) {
        book[i + 1] = book[i];               // slide it right
        --i;
        ++shifts;
    }
    book[i + 1] = o;
}

// Find a ticket in the sorted book. Returns the position, or -1 for absent.
int findTicket(const vector<Order>& book, int ticket, int& probes) {
    int left = 0, right = (int)book.size() - 1;
    probes = 0;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        ++probes;
        if (book[mid].ticket == ticket) return mid;
        if (book[mid].ticket > ticket) right = mid - 1;
        else                           left  = mid + 1;
    }
    return -1;
}
```

Listing 2.10 (Continued) *Profitina Laundry's counter book: incremental insertion to keep it ordered, binary search to find a ticket.*

```
int main() {
    // Tickets are handed out in order, but two customers came back to change
    // their service and were re-booked, so 1104 and 1109 arrive out of turn.
    vector<Order> arrivals = {
        {1101, 3.5, "reguler"}, {1102, 2.0, "kilat"},   {1103, 6.25, "reguler"},
        {1105, 1.5, "setrika"}, {1106, 4.0, "reguler"}, {1104, 8.0,  "kilat"},
        {1107, 2.75,"setrika"}, {1108, 5.0, "reguler"}, {1110, 3.0,  "kilat"},
        {1109, 7.5, "reguler"}
    };

    vector<Order> book;
    for (const Order& o : arrivals) insertOrder(book, o);

    cout << "the day's book, sorted by ticket:\n";
    double total = 0;
    for (const Order& o : book) {
        cout << "  " << o.ticket << "  " << fixed << setprecision(2)
             << setw(5) << o.kg << " kg  " << o.service << '\n';
        total += o.kg;
    }
    cout << "total weight: " << total << " kg\n";
    cout << "shifts done by all insertions: " << shifts
         << "  (worst case would be " << (arrivals.size() * (arrivals.size() - 1) / 2)
         << ")\n\n";

    for (int t : {1104, 1110, 1120}) {
        int probes;
        int at = findTicket(book, t, probes);
        cout << "collect ticket " << t << ": ";
        if (at < 0) cout << "not in today's book";
        else cout << book[at].kg << " kg " << book[at].service;
        cout << "  (" << probes << " probes)\n";
    }
    return 0;
}
```

Its actual output is:

```
the day's book, sorted by ticket:
  1101   3.50 kg  reguler
  1102   2.00 kg  kilat
  1103   6.25 kg  reguler
  1104   8.00 kg  kilat
  1105   1.50 kg  setrika
  1106   4.00 kg  reguler
  1107   2.75 kg  setrika
  1108   5.00 kg  reguler
  1109   7.50 kg  reguler
  1110   3.00 kg  kilat
total weight: 43.50 kg
shifts done by all insertions: 3  (worst case would be 45)

collect ticket 1104: 8.00 kg kilat  (4 probes)
collect ticket 1110: 3.00 kg kilat  (4 probes)
collect ticket 1120: not in today's book  (4 probes)
```

Two numbers in that output are the whole chapter in miniature.

Three shifts, against a worst case of 45. Ten orders arriving in a genuinely random order would cost about $n(n-1)/4 \approx 22$ shifts, and in the worst case 45 (Section 2.11.2). Because the tickets were nearly in order, we paid three. The best-case analysis of Section 2.11.1 is not a theoretical

curiosity — it is the case that actually occurs at a laundry counter, and it is why the simple algorithm is the right one here.

Four probes for ten orders. $\lg 10 \approx 3.3$, and the exact worst-case count derived in Solution 2.10, $\lceil \lg(n+1) \rceil$, gives 4 comparisons. Note that the *absent* ticket 1120 also took four probes, not ten: an unsuccessful binary search costs the same as a successful one, whereas an unsuccessful linear search is the maximum. With 5,000 tickets across a month the book would need at most 13 probes; linear search would need up to 5,000.

2.18 Common Mistakes, and How to Avoid Them

Confusing the algorithm with the program. "My algorithm is slow because C++ is slow" is almost never true. If your program is a thousand times too slow, you have an n^2 algorithm where an $n\lg n$ one is needed, and no language change will save you. *Fix:* count the operations before you blame the tooling.

Forgetting that a loop is not one step. The commonest analysis error by a wide margin: reading a function with a single nested loop and calling it $O(n)$. *Fix:* apply **Table 2.2** mechanically. Write the lines down, write the counts beside them, and multiply.

Reporting wall-clock time as if it were complexity. "It takes 0.4 seconds, so it's linear" says nothing. *Fix:* use the doubling test of Section 2.11.4. Double n and look at the ratio — 2 means linear, 4 means quadratic, a little over 2 means $n\lg n$.

Losing the key in Insertion Sort. Omitting `key = A[j]` and writing `A[i+1] = A[j]` at the end produces an array full of duplicated values, because the shifting has already overwritten $A[j]$. *Fix:* the invariant of Section 2.9.4 clause (c) names $A[i+1]$ as a *hole* — a hole only exists because you saved its contents first.

Inserting at $A[i]$ instead of $A[i+1]$. The inner loop always decrements i one step past the last element it moved. *Fix:* trace the loop's final iteration on paper once, as we did in Section 2.8.3, steps 14–16, and you will never write it wrong again.

Testing $A[i] > key$ before $i > 0$. This reads $A[0]$, outside the array. In C++ it may appear to work for months and then corrupt memory. *Fix:* the bounds test always goes first; `&&` short-circuits left to right.

Writing `mid = (left + right) / 2`. Correct in mathematics, overflowing in `int`, as **Listing 2.8** shows. *Fix:* `mid = left + (right - left) / 2`, always, as a reflex.

Writing `right = mid` instead of `right = mid - 1`. The window stops shrinking and the program hangs on a two-element window. *Fix:* after every binary search you write, ask "does the window strictly shrink in every branch?" If it can fail to, you have an infinite loop.

Running binary search on unsorted data. It returns NIL for values that are present, silently and confidently. *Fix:* the precondition belongs in a comment above the function and in the caller's mind; nothing in the code will check it for you.

Returning -1 for "not found" in a 1-based program, or 0 in a 0-based one. *Fix:* choose the sentinel that cannot possibly be a valid index in your scheme, and say which in a comment.

Treating "sorted" as free. Sorting to enable one binary search is slower than a single linear scan. *Fix:* Section 2.14.8 — count your queries before you buy the structure.

Optimizing constants when the growth rate is wrong. Rewriting a $\Theta(n^2)$ loop to be twice as fast turns sixteen minutes into eight. *Fix:* **Table 2.3**. Change the exponent, not the coefficient.

Proving correctness by testing. Ten passing test cases prove that ten inputs work. *Fix:* state the invariant, and check initialization, maintenance and termination. It takes fifteen minutes and it covers infinitely many inputs.

2.19 Chapter Summary

1. An **algorithm** is a step-by-step procedure taking input to output; a **program** is its implementation in a language; a **data structure** is how the data is organized. A **problem** specifies the input/output relation; an **instance** is one particular input. Correctness is a property of the algorithm across all instances; running time usually depends on the instance.
2. An algorithm must satisfy five properties: input, output, definiteness, finiteness, effectiveness. Being an algorithm is a low bar; being an efficient one is the goal.
3. The **five design goals** are correctness, efficiency, robustness, adaptability and reusability — the first two are design goals and are this book's subject, the last three are implementation goals belonging to software engineering. **Figure 2.16** is the map.
4. We analyze in the **RAM model**: one processor, uniform memory access, each simple instruction one step. It is a deliberate simplification, chosen because it is machine-independent and still predicts real performance.
5. To find $T(n)$: assign a constant cost to each line, count how many times each line runs, and sum the products. A loop is never one step.
6. **Insertion Sort** is correct, proved by the loop invariant "$A[1 \dots j-1]$ holds the original elements of $A[1 \dots j-1]$ in sorted order" via initialization, maintenance and termination.
7. Insertion Sort runs in $\Theta(n)$ in the best case (already sorted, $t_j = 1$), and $\Theta(n^2)$ in the worst (reverse sorted, $t_j = j$, so $\sum_{j=2}^{n} j = n(n+1)/2 - 1$). The average case is also $\Theta(n^2)$, at about half the worst-case constant.
8. We analyze the **worst case** because it is a guarantee, because adversarial and structured inputs are common, and because the average case usually has the same order anyway.
9. **Growth rate dominates everything** as n grows: $O(1) < O(\lg n) < O(\sqrt{n}) < O(n) < O(n\lg n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$. At $n = 10^6$, an $n\lg n$ algorithm takes milliseconds and an n^2 algorithm takes sixteen minutes.
10. **Linear search** is $\Theta(n)$ and optimal on unsorted data. **Binary search** is $\Theta(\lg n)$ on sorted data — an exponential improvement bought entirely by structure, and paid for once by an $\Theta(n\lg n)$ sort.
11. The **incremental** paradigm extends a partial solution one element at a time (Insertion Sort). **Divide and conquer** splits into sub-problems of the same kind (binary search, Merge Sort). **Ad hoc** solutions exploit structure peculiar to one problem.

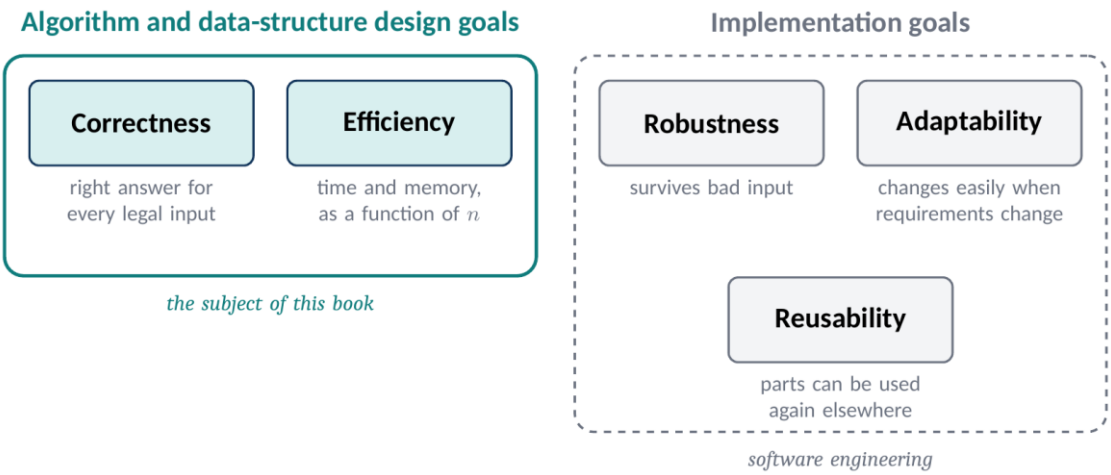


Figure 2.16 The five goals once more. Everything in this chapter served the two on the left — correctness, proved with loop invariants, and efficiency, derived by counting.

"An algorithm must be seen to be believed." — Donald Knuth [3]*. Trace every algorithm by hand. The understanding comes from the doing.*

2.20 Exercises

2.1. Trace Insertion Sort by hand on $A = \langle 31,41,59,26,41,58 \rangle$. Give the array after each outer iteration, and state t_j for each j.

2.2. For the array of Exercise 2.1, compute $\sum_{j=2}^{6} t_j$ and $\sum_{j=2}^{6}(t_j - 1)$. How do they compare with the best case $(n-1)$ and the worst case $(n(n+1)/2 - 1)$ for $n = 6$?

2.3. Rewrite **Listing 2.1** using 0-based indexing, with no unused element. State the new loop bounds and the new inner-loop test, and explain which comparison changes and why.

2.4. Modify Insertion Sort to sort into **non-increasing** order. Which single character changes? Write out the modified loop invariant.

2.5. Prove, using a loop invariant, that the procedure of **Pseudocode 2.5** correctly computes the sum of $A[1 \ldots n]$.

Pseudocode 2.5 Summing the elements of an array.

```
SUM(A, n)
s = 0
for i = 1 to n
    s = s + A[i]
return s
```

2.6. An **inversion** of an array A is a pair of indices (i, j) with $i < j$ and $A[i] > A[j]$. List all the inversions of $\langle 2,3,8,6,1 \rangle$. What is the maximum possible number of inversions in an array of n distinct elements, and which array achieves it?

2.7. Prove that the total number of shifts performed by Insertion Sort — that is, $\sum_{j=2}^{n}(t_j - 1)$ — equals exactly the number of inversions in the input. Use this to give a second derivation of the worst-case running time.

2.8. Write a C++ program that verifies the claim of Exercise 2.7 empirically on random arrays.

2.9. Insertion Sort finds the insertion point by scanning linearly. Since $A[1 \dots j-1]$ is sorted, we could instead find the point by binary search. Does this reduce the worst-case running time of the sort? Justify your answer by counting comparisons and movements separately.

2.10. Give the exact number of comparisons that binary search makes, in the worst case, on an array of n elements, for $n = 1,2,3, \dots ,8$. Express the general answer as a formula.

2.11. The array $A = \langle 1,3,3,3,3,7,9,9,11 \rangle$ contains duplicates. Running **Listing 2.7** with $q = 3$ returns *some* index holding a 3, but which one is not specified. Write a variant that returns the **first** occurrence, and one that returns the **last**, both still in $\Theta(\lg n)$.

2.12. A student writes `while (left < right)` instead of `while (left <= right)` in binary search. Give a concrete array and query for which this version returns the wrong answer, and trace it to show why.

2.13. You must decide between linear search and "sort once, then binary search" for an array of $n = 2^{20}$ elements. Derive the number of queries k at which the second becomes cheaper, assuming the sort costs $n\lg n$ and each binary search costs $\lg n$.

2.14. A problem statement guarantees $1 \le n \le 200{,}000$ and allows two seconds. Which complexities from **Table 2.4** are acceptable? If your first idea is an $O(n^2)$ algorithm, how many operations would it perform at the maximum n, and by what factor does it exceed the budget?

2.15. Insertion Sort is $\Theta(n^2)$ and Merge Sort is $\Theta(n\lg n)$, yet library sorts use Insertion Sort for small subarrays. Suppose Insertion Sort costs $8n^2$ operations and Merge Sort costs $64n\lg n$. For which values of n does Insertion Sort win?

2.21 Solutions to the Exercises

Solution 2.1. The array is $\langle 31,41,59,26,41,58 \rangle$, $n = 6$.

- $j = 2$: $key = 41$. Test $A[1] = 31 > 41$? No. So $t_2 = 1$, no shifts. Array: $\langle 31,41,59,26,41,58 \rangle$.
- $j = 3$: $key = 59$. Test $A[2] = 41 > 59$? No. $t_3 = 1$. Array unchanged.
- $j = 4$: $key = 26$. $A[3] = 59 > 26$ yes, shift. $A[2] = 41 > 26$ yes, shift. $A[1] = 31 > 26$ yes, shift. $i = 0$, test fails. Four tests, so $t_4 = 4$, three shifts. Array: $\langle 26,31,41,59,41,58 \rangle$.
- $j = 5$: $key = 41$. $A[4] = 59 > 41$ yes, shift. $A[3] = 41 > 41$? No — the test is strict, so equal elements stop the scan. Two tests, $t_5 = 2$, one shift. Array: $\langle 26,31,41,41,59,58 \rangle$.
- $j = 6$: $key = 58$. $A[5] = 59 > 58$ yes, shift. $A[4] = 41 > 58$? No. $t_6 = 2$, one shift. Array: $\langle 26,31,41,41,58,59 \rangle$.

Final: $\langle 26,31,41,41,58,59 \rangle$. Note in $j = 5$ that the strict `>` is what keeps the two 41s in their original relative order — that property is called **stability**, and it is why `>` and not `>=` is written on line 4.

Solution 2.2. $\sum t_j = 1 + 1 + 4 + 2 + 2 = 10$ and $\sum(t_j - 1) = 0 + 0 + 3 + 1 + 1 = 5$. For $n = 6$ the best case gives $\sum t_j = n - 1 = 5$ and the worst gives $n(n+1)/2 - 1 = 20$. Our value of 10 sits between them and close to the average-case prediction of about half the worst case, which is 10. The input was moderately disordered.

Solution 2.3. With 0-based indices the sorted prefix is $A[0 \dots j-1]$. **Listing 2.11** is the complete program, sorting the same array as **Listing 2.1**.

Listing 2.11 Insertion Sort with ordinary 0-based indexing and no unused element.

```
// Exercise 2.3: Insertion Sort with ordinary 0-based indexing.
// The sorted prefix is now A[0..j-1], and index 0 is a real position.
#include <iostream>
#include <vector>
using namespace std;

void insertionSort0(vector<int>& A) {
    int n = (int)A.size();
    for (int j = 1; j < n; ++j) {          // was: j = 2 to n
        int key = A[j], i = j - 1;
        while (i >= 0 && A[i] > key) {     // was: i > 0
            A[i + 1] = A[i]; --i;
        }
        A[i + 1] = key;
    }
}

int main() {
    vector<int> A = {5, 2, 4, 6, 1, 3};    // no unused element this time
    insertionSort0(A);
    for (size_t i = 0; i < A.size(); ++i) cout << A[i] << (i + 1 < A.size() ? ' ' : '\n');
    return 0;
}
```

Compiled with `g++ -std=c++17 -O2 -Wall` and run, it prints:

```
1 2 3 4 5 6
```

Two changes. The outer loop starts at $j = 1$ (the second element is now at index 1, not 2) and runs while $j < n$. The inner guard becomes $i \geq 0$ rather than $i > 0$, because index 0 is now a legal position that must be examined. Everything else, including `A[i+1] = key`, is unchanged — which is a useful sign that the logic, not the indexing, is what the invariant captured.

Solution 2.4. Change `A[i] > key` to `A[i] < key` on line 4: one character. The invariant becomes "at the start of each iteration, $A[1 \ldots j-1]$ consists of the elements originally in $A[1 \ldots j-1]$ in **non-increasing** order", and every step of the proof in Section 2.9 goes through with the inequality reversed.

Solution 2.5. *Invariant:* at the start of each iteration of the loop of line 2, $s = \sum_{k=1}^{i-1} A[k]$.

Initialization: before the first iteration $i = 1$, so the claim is $s = \sum_{k=1}^{0} A[k]$, an empty sum, which is 0 by definition. Line 1 set $s = 0$. ✓

Maintenance: assume $s = \sum_{k=1}^{i-1} A[k]$. Line 3 sets $s \leftarrow s + A[i] = \sum_{k=1}^{i-1} A[k] + A[i] = \sum_{k=1}^{i} A[k]$. The loop then increments i, so with the new value of i the invariant again reads $s = \sum_{k=1}^{i-1} A[k]$. ✓

Termination: the loop ends with $i = n + 1$, so $s = \sum_{k=1}^{n} A[k]$, which is the sum of the whole array. Line 4 returns it. ✓ ■

Solution 2.6. For $\langle 2,3,8,6,1 \rangle$ with 1-based indices, the inversions are $(1,5)$ since $2 > 1$; $(2,5)$ since $3 > 1$; $(3,4)$ since $8 > 6$; $(3,5)$ since $8 > 1$; and $(4,5)$ since $6 > 1$. That is **five** inversions.

The maximum is $\binom{n}{2} = \frac{n(n-1)}{2}$, achieved by a strictly decreasing array: every one of the $\binom{n}{2}$ pairs $i < j$ then has $A[i] > A[j]$, and no array can have more pairs than it has pairs.

Solution 2.7. Each execution of line 5 moves one element exactly one position to the right, past the key. An element $A[i]$ is moved past $key = A[j]$ precisely when $i < j$ and $A[i] > key$ — that is, precisely when (i, j) is an inversion. Conversely, every inversion (i, j) causes exactly one shift: when the outer loop reaches j, the element originally at i is still to the left of the key's landing

place and greater than it, so it is shifted once and never again. The correspondence is one-to-one, so the total number of shifts equals the number of inversions.

Second derivation of the worst case: by Exercise 2.6 the maximum number of inversions is $n(n-1)/2$, attained by a reverse-sorted array. So the maximum number of shifts is $n(n-1)/2 = \Theta(n^2)$, and since each shift costs $\Theta(1)$ and the remaining work is $\Theta(n)$, the worst-case running time is $\Theta(n^2)$ — the same conclusion as Section 2.11.2, reached without summing a series.

Solution 2.8.

Listing 2.12 Verifying that Insertion Sort's shift count equals the input's inversion count.

```
// Exercise: the number of shifts Insertion Sort performs is exactly the number
// of inversions in the input. This program counts both and compares them.
// An inversion is a pair of positions (i, j) with i < j and A[i] > A[j].
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
using namespace std;

long long countInversions(const vector<int>& A, int n) {
    long long inv = 0;
    for (int i = 1; i <= n; ++i)
        for (int j = i + 1; j <= n; ++j)
            if (A[i] > A[j]) ++inv;          // the brute-force definition
    return inv;
}

long long insertionShifts(vector<int> A, int n) {   // by value: we modify a copy
    long long shifts = 0;
    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; ++shifts; }
        A[i + 1] = key;
    }
    return shifts;
}

int main() {
    vector<int> demo = {0, 2, 3, 8, 6, 1};              // the CLRS example
    cout << "A = [2,3,8,6,1]: inversions = " << countInversions(demo, 5)
         << ", shifts = " << insertionShifts(demo, 5) << '\n';

    mt19937 rng(2024);
    bool allEqual = true;
    for (int trial = 0; trial < 200; ++trial) {
        int n = 1 + (int)(rng() % 40);
        vector<int> A(n + 1);
        for (int i = 1; i <= n; ++i) A[i] = (int)(rng() % 100);
        if (countInversions(A, n) != insertionShifts(A, n)) allEqual = false;
    }
    cout << "200 random trials agree: " << (allEqual ? "yes" : "no") << '\n';
    return 0;
}
```

The output of **Listing 2.12** is:

```
A = [2,3,8,6,1]: inversions = 5, shifts = 5
200 random trials agree: yes
```

The five inversions found by the program are the five listed by hand in Solution 2.6. Note that the arrays in the random trials contain duplicates, and the identity still holds — because the strict `>` in both the inversion definition and the loop test treats equal elements the same way.

Solution 2.9. No: the worst case remains $\Theta(n^2)$.

Count the two kinds of work separately. **Comparisons** do fall: binary search locates the insertion point among $j-1$ sorted elements using $\Theta(\lg j)$ comparisons instead of up to j, so the total drops from $\Theta(n^2)$ to $\sum_{j=2}^{n}\Theta(\lg j) = \Theta(n\lg n)$. But **movements** do not change at all. Once the insertion point is known, every element to its right must still be shifted one place, and in the worst case that is $j-1$ movements for each j, totalling $n(n-1)/2 = \Theta(n^2)$ as before.

The running time is the sum of both, so it stays $\Theta(n^2)$. The lesson is general and important: speeding up one part of an algorithm changes nothing if another part has the same order. This variant is worth having only when comparisons are far more expensive than moves — comparing long strings, say, while moving only pointers.

Solution 2.10. Let $C(n)$ be the worst-case number of comparisons on n elements. Each comparison reduces the window from n to at most $\lfloor n/2 \rfloor$:

Table 2.8 Worst-case comparison counts for binary search on n elements.

n	1	2	3	4	5	6	7	8
$C(n)$	1	2	2	3	3	3	3	4

Reading **Table 2.8**, $C(n)$ steps up by one exactly at $n = 1,2,4,8$ — the powers of two. The formula is $C(n) = \lfloor \lg n \rfloor + 1$, equivalently $\lceil \lg(n+1) \rceil$. Check: $n = 5$ gives $\lfloor \lg 5 \rfloor + 1 = 2 + 1 = 3$ ✓, and $n = 10^6$ gives $\lfloor 19.93 \rfloor + 1 = 20$ ✓, which is the figure quoted in Section 2.14.5.

Solution 2.11. The fix is to *not* return immediately on a hit. Record the position, then keep searching the side where an earlier (or later) occurrence could still lie.

Listing 2.13 Binary search for the first and last occurrence of a value, with duplicates present.

```
// Exercise: find the FIRST occurrence of q in a sorted array with duplicates.
// Plain binary search may land in the middle of a run of equal values, so the
// loop must keep going left after a hit instead of returning at once.
#include <iostream>
#include <vector>
using namespace std;

// Returns the smallest index i (0-based) with A[i] == q, or -1 if absent.
int firstOccurrence(const vector<int>& A, int q) {
    int left = 0, right = (int)A.size() - 1, answer = -1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (A[mid] == q) { answer = mid; right = mid - 1; }  // remember, keep going left
        else if (A[mid] > q) right = mid - 1;
        else                 left  = mid + 1;
    }
    return answer;
}

// Returns the largest index i (0-based) with A[i] == q, or -1 if absent.
int lastOccurrence(const vector<int>& A, int q) {
    int left = 0, right = (int)A.size() - 1, answer = -1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (A[mid] == q) { answer = mid; left = mid + 1; }   // keep going right
        else if (A[mid] > q) right = mid - 1;
        else                 left  = mid + 1;
    }
    return answer;
}

int main() {
    vector<int> A = {1, 3, 3, 3, 3, 7, 9, 9, 11};
    for (int q : {3, 9, 5}) {
        int f = firstOccurrence(A, q), l = lastOccurrence(A, q);
        cout << "q=" << q << ": first=" << f << " last=" << l
             << " count=" << (f < 0 ? 0 : l - f + 1) << '\n';
    }
    return 0;
}
```

The output of **Listing 2.13** is:

```
q=3: first=1 last=4 count=4
q=9: first=6 last=7 count=2
q=5: first=-1 last=-1 count=0
```

Both variants remain $\Theta(\lg n)$: the window still at least halves every iteration, since `right = mid - 1` and `left = mid + 1` both exclude mid. The extra work is that we never exit early, so we always pay the full $\lfloor \lg n \rfloor + 1$ comparisons rather than sometimes fewer — a constant-factor cost for an exact answer. Subtracting the two results gives the count of occurrences in $\Theta(\lg n)$, which is how one counts duplicates in a sorted array without scanning them. These are the routines the C++ standard library calls `std::lower_bound` and `std::upper_bound` [4].

Solution 2.12. Take $A = \langle 5 \rangle$ with $n = 1$ and $q = 5$, using 1-based indices.

With `while (left < right)`: initially $left = 1$, $right = 1$. The test $1 < 1$ is **false**, so the loop body never runs and the function returns NIL — even though $A[1] = 5 = q$. Wrong answer.

The bug is not confined to $n = 1$. Take $A = \langle 2,4 \rangle$ and $q = 4$: $left = 1$, $right = 2$, test $1 < 2$ true, $mid = 1$, $A[1] = 2 < 4$, so $left = 2$. Now the test $2 < 2$ is false and the loop exits with NIL, although $A[2] = 4$. In general `<` fails whenever the answer is the last element still in the window, because a one-element window is never examined. The `<=` version examines it and is correct —

this is exactly the class of off-by-one that Bentley reports as afflicting most programmers' first attempt at binary search [2].

Solution 2.13. With $n = 2^{20} = 1{,}048{,}576$ we have $\lg n = 20$.

- Linear search: k queries cost $kn = 1{,}048{,}576\,k$.
- Sort then binary search: $n\lg n + k\lg n = 20{,}971{,}520 + 20k$.

Set them equal:

$$1{,}048{,}576\,k = 20{,}971{,}520 + 20k \;\Rightarrow\; 1{,}048{,}556\,k = 20{,}971{,}520 \;\Rightarrow\; k \approx 20.0.$$

So from about 20 queries onward, sorting pays. In general the break-even point is

$$k \approx \frac{n\lg n}{n - \lg n} \approx \lg n, \tag{2.13}$$

and Equation (2.13) is a pleasing rule: **sorting is worth it once you expect more than about** $\lg n$ **searches.** For a million elements that is twenty searches — a very low bar, which is why in practice one almost always sorts.

Solution 2.14. From **Table 2.4**, $n \le 200{,}000$ sits in the $O(n\lg n)$ band, so $O(n\lg n)$, $O(n)$, $O(\lg n)$ and $O(1)$ are acceptable; $O(n^2)$ is not.

An $O(n^2)$ algorithm at $n = 200{,}000$ performs $(2 \times 10^5)^2 = 4 \times 10^{10}$ operations. At an optimistic 10^9 operations per second that is 40 seconds against a 2-second limit: **20 times over budget**. No constant-factor optimization recovers a factor of twenty reliably. By contrast $O(n\lg n)$ gives $2 \times 10^5 \times 17.6 \approx 3.5 \times 10^6$ operations — a few milliseconds, with four orders of magnitude to spare. This is the calculation to perform *before* writing code, as the Five-Phase Framework of Chapter 1 and the constraint tables of CP4 [5] both insist.

Solution 2.15. Insertion Sort wins when $8n^2 < 64n\lg n$. Divide both sides by $8n$, which is positive:

$$n < 8\lg n.$$

Solve by inspection, since the equation $n = 8\lg n$ has no closed form:

- $n = 8$: is $8 < 8 \times 3 = 24$? Yes.
- $n = 16$: is $16 < 8 \times 4 = 32$? Yes.
- $n = 32$: is $32 < 8 \times 5 = 40$? Yes.
- $n = 43$: $\lg 43 \approx 5.43$, so $8\lg n \approx 43.4$. Is $43 < 43.4$? Yes, barely.
- $n = 44$: $\lg 44 \approx 5.46$, so $8\lg n \approx 43.7$. Is $44 < 43.7$? No.

So Insertion Sort is faster for $n \le 43$ and Merge Sort from $n \ge 44$. That is exactly why production sorts are hybrids: recurse with the asymptotically better algorithm until the subarray is small, then switch to the one with the better constant. Real implementations use a cutoff in the range 16 to 32, a little below our idealized 43 because real constants are not the round numbers of this exercise [4].

Further reading

Chapters 1 to 4 of CLRS [1] cover this material in its canonical form and should be your first stop; Knuth's Volume 1 [3] is the source for the five properties of an algorithm, and Volume 3 [6] contains the definitive treatment of insertion sorting and of binary search, including the observation that although the idea was published in 1946, the first published *correct* implementation for arrays of arbitrary length did not appear until 1962. Bentley [2] tells the story of how hard the latter really is. For the competitive-programming view of the constraint tables in Section 2.12.5, see CP4 [5].

References

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

[2] J. Bentley, *Programming Pearls*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2000.

[3] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[4] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[5] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, 4th ed. Raleigh, NC, USA: Lulu Press, 2020.

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

Chapter 3

Asymptotic Notation, Recurrences, and the Master Theorem

By the end of this chapter you will be able to take any algorithm that calls itself, write down the equation that describes its running time, and solve that equation three different ways — by guessing and proving, by drawing a tree, and by a recipe that answers most such equations in about thirty seconds. You will know exactly what O, Ω, Θ, o and ω mean, not as vague slogans but as statements about constants you can name and write down. You will be able to prove that a bound holds, and — just as important — prove that a bound does *not* hold. You will be able to state the Master Theorem from memory, apply all three of its cases, check its regularity condition, and recognize the two gaps where it silently refuses to answer. And you will have derived Merge Sort from nothing, proved its merge step correct with a loop invariant, and shown that it runs in $\Theta(n\lg n)$ time.

3.1 Where We Left Off

Chapter 2 gave us two things: a machine to count on, and a method to trust.

The machine was the **RAM model** — the random-access machine. It is a pretend computer with a fixed set of simple instructions (add, subtract, multiply, compare, read a memory cell, write a memory cell, jump), and every one of those instructions takes one unit of time. There is no cache, no pipeline, no operating system stealing cycles. We count instructions, not seconds. The reason is not laziness; it is that seconds are a property of a particular laptop on a particular Tuesday, and instruction counts are a property of the algorithm itself. If your becak-repair shop replaces its ten-year-old computer with a new one that is four times faster, every algorithm you own gets four times faster, and nothing you learned about which algorithm is better changes at all.

The method was the **loop invariant**. An algorithm is *correct* if, for every legal input, it terminates and produces the desired output. We proved Insertion Sort correct by finding a statement — "at the start of each iteration of the outer loop for index j, the subarray $A[1..j-1]$ holds the elements originally in $A[1..j-1]$, but in sorted order" — and then showing three things about it:

1. **Initialization.** The statement is true before the first iteration. When $j = 2$, the subarray $A[1..1]$ is one element long, and one element is trivially in sorted order.
2. **Maintenance.** If the statement is true before an iteration, it is still true before the next one. The inner `while` loop shifts $A[j-1], A[j-2], \ldots$ one position to the right, without disturbing their order among themselves, until it finds the slot where the key belongs, and drops the key in. What was sorted in $A[1..j-1]$ is now sorted in $A[1..j]$.
3. **Termination.** When the loop finally stops — at $j = n + 1$ — the statement says "$A[1..n]$ holds the original elements in sorted order", which is precisely what we wanted.

That is mathematical induction wearing different clothes. Initialization is the base case, maintenance is the inductive step, termination is the conclusion you cash in at the end.

3.1.1 Partial correctness and total correctness

Chapter 2 proved that Insertion Sort *works*. It did not separate the two different promises hidden inside that word, and the difference between them matters the moment we start writing algorithms that call themselves. **Figure 3.1** draws the distinction in a single picture.

> **DEFINITION.** An algorithm is **partially correct** if, *whenever it terminates*, it produces the correct output. Partial correctness says nothing at all about whether the algorithm ever terminates.
>
> **DEFINITION.** An algorithm is **totally correct** if (1) it always terminates on every legal input, *and* (2) when it terminates it produces the correct output. In short: **total correctness = partial correctness + termination.**

Partial correctness is the promise "*if* the chef finishes cooking, the dish will be delicious." It is a real promise and a useful one, but it does not stop the chef standing in the kitchen forever. Total correctness adds the second promise: the chef finishes.

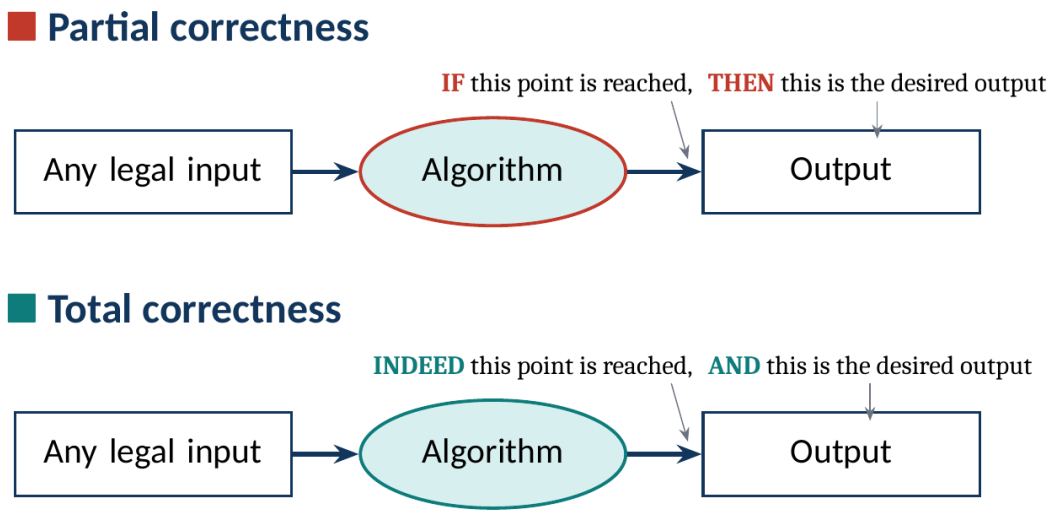


Figure 3.1 Partial correctness against total correctness. In the upper diagram the guarantee is conditional — **IF** control reaches the output box, **THEN** the output is the desired one. In the lower diagram the guarantee is unconditional — control **INDEED** reaches the output box, **AND** the output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

Why can we not simply write a program that reads any algorithm and decides whether it is correct? Because no such program can exist. The question "does this algorithm halt on this input?" is the **Halting Problem**, and Alan Turing proved in 1936 that no algorithm decides it in general. Correctness is therefore something we prove by hand, one algorithm at a time, with tools like loop invariants and induction. It is not something we can automate away.

For most algorithms in this book, termination is obvious — a loop counter climbs toward a fixed limit, or a recursive call is made on a strictly smaller array — and the hard, interesting work is proving partial correctness.

3.1.2 What is still missing

Chapter 2 left three holes, and this chapter fills all three.

The first hole is **language**. Chapter 2 introduced the growth of functions informally and said Insertion Sort takes "about n^2 steps in the worst case" and "about n steps in the best case". What does "about" mean, exactly? If one implementation takes $3n^2 + 5n + 7$ steps and another takes $100n^2$ steps, are they the same or different? We need definitions precise enough to argue with, not adjectives — definitions with constants in them that you can write down and check.

The second hole is **arithmetic**. Counting the work of nested loops produces sums — $1 + 2 + \cdots + (n - 1)$, or $1 + \frac{3}{4} + \frac{9}{16} + \cdots$ — and we need closed forms for those sums, with the derivations, not just the answers.

The third hole is **recursion**. Insertion Sort is a loop, so a sum describes it. But Merge Sort calls itself, twice, on halves. No sum describes that directly. It needs a *recurrence*: an equation that defines a function in terms of itself on smaller inputs. Most of this chapter is about solving those.

3.2 Why We Need a Language for Growth

Suppose you time two sorting programs on your own machine and get this:

- Program A sorts 1,000 records in 0.8 seconds.
- Program B sorts 1,000 records in 0.1 seconds.

Program B is eight times faster, so B wins. Now run both on 1,000,000 records. A takes about 27 minutes. B takes about 28 hours. B lost, and it lost badly, because A's running time grows like $n\lg n$ and B's grows like n^2, and at $n = 1{,}000{,}000$ that difference swamps the factor of eight. Here is the arithmetic. Multiplying n by 1000 multiplies $n\lg n$ by $1000 \times \frac{\lg 10^6}{\lg 10^3} = 1000 \times 2 = 2000$, so A needs $0.8 \times 2000 = 1600$ seconds, about 27 minutes. The same step multiplies n^2 by $1000^2 = 10^6$, so B needs $0.1 \times 10^6 = 100{,}000$ seconds, about 28 hours.

This is the whole motivation for asymptotic analysis in one example. The **constant factor** — the 8 — is a property of the machine, the compiler, the programmer's care with memory layout, and the phase of the moon. The **growth rate** — $n\lg n$ against n^2 — is a property of the algorithm, and it is the thing that decides who wins on real data.

> **KEY INSIGHT.** Asymptotic analysis throws away exactly the information that depends on the machine, and keeps exactly the information that depends on the algorithm. Two rules do all the throwing away: **drop constant factors** ($3n^2 \to n^2$) and **drop lower-order terms** ($5n^2 + 3n + 7 \to n^2$). What is left is how the running time grows as the input grows without bound.

3.2.1 The numbers, so you believe it

Slogans are cheap. **Listing 3.1** prints the actual values of the growth functions we will meet, so that the reader can see for themselves how differently they behave.

Listing 3.1 A C++ program that tabulates the standard growth functions. Save it as `growth.cpp`, compile with `g++ -std=c++17 -O2 -o growth growth.cpp`, and run `./growth`.

```
// growth.cpp - How fast do the standard growth functions really grow?
// Compile: g++ -std=c++17 -O2 -o growth growth.cpp
#include <iostream>
#include <iomanip>
#include <string>
#include <cmath>
#include <sstream>
using namespace std;

// Return a human-readable form: exact when small, scientific when huge.
static string pretty(double v) {
    if (v < 0) return "overflow";
    ostringstream os;
    if (v < 1e15) { os << fixed << setprecision(0) << v; }
    else          { os << scientific << setprecision(2) << v; }
    return os.str();
}

int main() {
    const long long sizes[] = {8, 16, 32, 64, 1024, 1048576};
    cout << left << setw(10) << "n"
         << right << setw(10) << "lg n"
         << setw(14) << "n"
         << setw(16) << "n lg n"
         << setw(18) << "n^2"
         << setw(14) << "2^n" << "\n";
    cout << string(82, '-') << "\n";
    for (long long n : sizes) {
        double lg = log2((double)n);
        // 2^n is astronomically large past n = 64, so print it in scientific form.
        double pow2 = (n <= 1024) ? pow(2.0, (double)n) : INFINITY;
        cout << left << setw(10) << n
             << right << setw(10) << pretty(lg)
             << setw(14) << pretty((double)n)
             << setw(16) << pretty(n * lg)
             << setw(18) << pretty((double)n * (double)n)
             << setw(14) << (isinf(pow2) ? string("astronomical") : pretty(pow2)) << "\n";
    }
    return 0;
}
```

Running it produces exactly this:

```
n               lg n             n          n lg n               n^2           2^n
----------------------------------------------------------------------------------
8                   3             8              24                64           256
16                  4            16              64               256         65536
32                  5            32             160              1024    4294967296
64                  6            64             384              4096      1.84e+19
1024               10          1024           10240           1048576  astronomical
1048576            20       1048576        20971520     1099511627776  astronomical
```

Read the last row. At one million records a lg$n$ algorithm does about 20 units of work; an n algorithm does one million; an $n$lg$n$ algorithm does twenty-one million; an n^2 algorithm does a *million million*. A machine that performs a billion units of work per second finishes the $n$lg$n$ job in about one fiftieth of a second and the n^2 job in about eighteen minutes. And 2^n at $n = 64$ is already 1.84×10^{19}, which at a billion per second is about 1.84×10^{10} seconds — nearly six hundred years.

Table 3.1 summarizes the same story in words, with the name each growth class goes by and an algorithm from this book that belongs to it.

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

The "doubling" column is the one to memorize, because it is the one you can feel. If an algorithm is quadratic, then going from 10,000 customers to 20,000 customers does not make the nightly job twice as slow — it makes it four times as slow. That is the difference between a job that finishes before the shop opens and one that does not.

A caution about small inputs. Asymptotic analysis describes behavior as n grows *without bound*. For small n the constants can dominate completely: Insertion Sort, which is $\Theta(n^2)$, genuinely beats Merge Sort, which is $\Theta(n\lg n)$, on arrays of ten or twenty elements, because its constant factor is tiny and it touches memory in a friendly pattern. Real library sort routines exploit this — they run Merge Sort or Quicksort down to blocks of about 16 elements and then finish with Insertion Sort. Asymptotics tell you which algorithm wins *eventually*; measurement tells you where "eventually" starts.

3.3 Big-O: The Asymptotic Upper Bound

We now make "grows no faster than" precise. Before the definition, here is every symbol it uses, in words.

- n is the **size of the input** — the number of records to sort, the number of characters in a string, the number of web pages to score. It is a non-negative integer.
- $f(n)$ is the **function we are studying**, usually the running time of an algorithm measured in RAM-model steps.
- $g(n)$ is the **simple function we are comparing against** — something like n, n^2, or $n\lg n$.
- c is a **positive constant multiplier**. It is a single fixed number, chosen once, that does not change as n changes. It absorbs every machine-dependent factor.
- n_0 is a **threshold**, a fixed non-negative integer. We promise nothing about what happens below it; we promise everything about what happens at and above it.

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

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

In plain words, Equation (3.1) says: f **eventually stays under some fixed multiple of g, and stays there forever.** You are allowed to pick the multiple. You are allowed to ignore the small inputs. You are not allowed to change your choice as n grows.

Figure 3.2 is the picture. The thick curve is $f(n)$. The thin curve is $c \cdot g(n)$ — that is g stretched vertically by the constant c you chose. To the left of n_0 the thick curve wanders wherever it likes

and may sit above the thin one; nobody cares. From n_0 rightwards, the thick curve is trapped underneath.

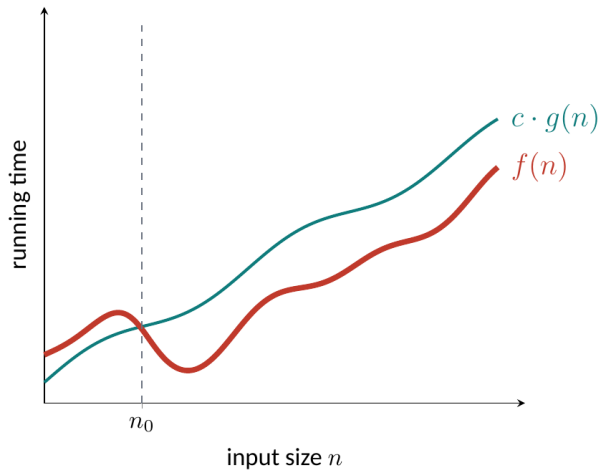


Figure 3.2 Big-O as an upper bound. Beyond the threshold n_0, the running time $f(n)$ never rises above $c \cdot g(n)$. What happens to the left of n_0 is irrelevant to the claim.

Big-O is what we use for **worst-case upper bounds**: "this algorithm will never take more than about this much time."

3.3.1 A proof, done completely: $7n - 3 = O(n)$

This is the classic first example, and it is the right first one because the whole proof fits in two lines — but we will spend a page on those two lines, because the *shape* of the argument is what matters.

Claim. $7n - 3 = O(n)$.

What we must produce. Two numbers: a constant $c > 0$ and a threshold n_0. Then we must show the inequality $7n - 3 \leq c \cdot n$ holds for every $n \geq n_0$. That is all a proof of an O-claim ever is: exhibit the witnesses, verify the inequality.

Choosing the witnesses. Look at $7n - 3$. It is $7n$ with something subtracted. Subtracting makes things smaller. So $7n - 3 \leq 7n$ already, for every n. That suggests $c = 7$.

The verification, one step per line.

$$7n - 3 \leq 7n \qquad \text{because } -3 \leq 0$$

$$7n = 7 \cdot n \qquad \text{so we may take } c = 7$$

and the inequality holds for every $n \geq 1$, so $n_0 = 1$.

Therefore $7n - 3 \leq 7n$ for all $n \geq 1$, with $c = 7$ and $n_0 = 1$, and by the definition $7n - 3 = O(n)$. ■

Notice what we did *not* have to do. We did not have to find the smallest c. We did not have to find the smallest n_0. Any pair that works is a complete proof. $c = 100$ and $n_0 = 50$ would also have worked; the claim would be no weaker.

3.3.2 A proof where the threshold does real work: $3n^2 + 5n + 7 = O(n^2)$

Here the choice $c = 3$ will not do, because $3n^2 + 5n + 7$ is strictly larger than $3n^2$ for every n. We must pay for the lower-order terms out of the constant.

Step 1 — bound each term by a multiple of n^2. For $n \geq 1$ we have $n \leq n^2$ and $1 \leq n^2$. Therefore

$$3n^2 + 5n + 7 \leq 3n^2 + 5n^2 + 7n^2 = 15n^2 \qquad \text{for all } n \geq 1.$$

So $c = 15$, $n_0 = 1$ is already a valid witness pair, and we are done. But it is instructive to ask how small c can be.

Step 2 — pushing c down. Try $c = 4$. We need $3n^2 + 5n + 7 \leq 4n^2$, i.e. $5n + 7 \leq n^2$. At $n = 6$: $37 \leq 36$ is false. At $n = 7$: $42 \leq 49$ is true, and since n^2 pulls ahead of $5n + 7$ from then on, the inequality holds for all $n \geq 7$. So $c = 4$, $n_0 = 7$ also works.

This is the trade the threshold exists to allow: a smaller constant bought with a larger threshold. **Listing 3.2** performs exactly this search mechanically.

3.3.3 A disproof: $n^2 \neq O(n)$

Proving that something is *not* O of something else is a different kind of argument, and students who have never seen one written out tend to flounder. The structure is **proof by contradiction**: assume witnesses exist, then derive an impossibility.

Claim. $n^2 \neq O(n)$.

Proof. Suppose, for contradiction, that $n^2 = O(n)$. Then by the definition there are positive constants c and n_0 with

$$n^2 \leq c \cdot n \qquad \text{for all } n \geq n_0.$$

Since $n \geq n_0 \geq 0$ and we may assume $n \geq 1$, divide both sides by n (a positive quantity, so the inequality direction is preserved):

$$n \leq c \qquad \text{for all } n \geq n_0.$$

But this says the integer n never exceeds the fixed number c, no matter how large n gets. Take $n = \lfloor c \rfloor + 1$, which is certainly at least n_0 if we also take n at least n_0 — concretely, take $n = \max(n_0, \lfloor c \rfloor + 1)$. For that n we have $n > c$, contradicting $n \leq c$.

The assumption must be false, so $n^2 \neq O(n)$. ■

> **KEY INSIGHT.** To *prove* $f = O(g)$, exhibit one pair (c, n_0) that works. To *disprove* it, show that *every* pair fails — usually by assuming a pair exists and manufacturing an n beyond n_0 that breaks the inequality. Proving is existential; disproving is universal. That asymmetry is why disproofs feel harder.

3.3.4 Letting the machine hunt for witnesses

A proof is a proof, but it helps enormously to *see* the constants. **Listing 3.2** searches for the smallest integer c, and then the smallest matching n_0, that witness a claim on a finite range — and reports honestly when no witness turns up.

Listing 3.2 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;
}
```

The program prints:

```
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
```

The first line is our hand proof, confirmed. The second line found $c = 4, n_0 = 7$ — exactly the pair we computed in §3.3.2. The third shows the trade again: $2n + 10 \leq 3n$ needs $10 \leq n$, so $n_0 = 10$. The last two lines are a warning as much as a result: **a program can only ever suggest that a claim is false, never prove it**, because it checks finitely many n. The proof in §3.3.3 is what settles the matter.

3.4 Big-Ω: The Asymptotic Lower Bound

Everything reverses.

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

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

In plain words, Equation (3.2) says: f **eventually stays above some fixed multiple of g, and stays there forever.** The constant c and the threshold n_0 mean exactly what they meant before; only the direction of the inequality has flipped.

Figure 3.3 draws it. The thick curve $f(n)$ dips below the thin curve $c \cdot g(n)$ for small n, which is permitted, and then climbs above it at n_0 and never comes back down.

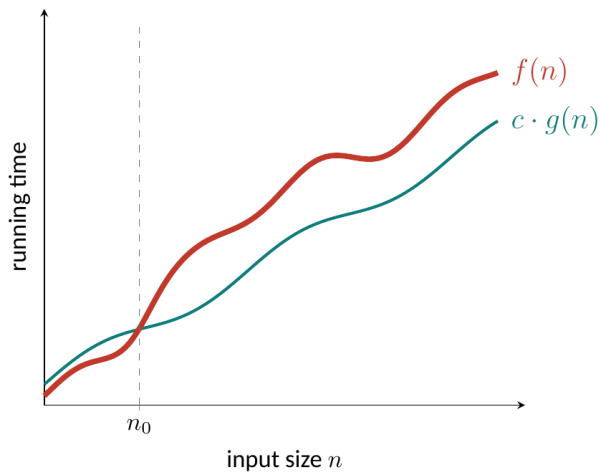


Figure 3.3 Big-Ω as a lower bound. Beyond n_0, the running time $f(n)$ never falls below $c \cdot g(n)$.

Worked example. Show $7n - 3 = \Omega(n)$.

We need $c \cdot n \le 7n - 3$ for large n. Try $c = 1$:

$$n \le 7n - 3 \Leftrightarrow 3 \le 6n \Leftrightarrow n \ge \frac{1}{2}.$$

So $c = 1$ and $n_0 = 1$ work, and $7n - 3 = \Omega(n)$. ■

A more ambitious c: try $c = 6$. We need $6n \le 7n - 3$, i.e. $3 \le n$, so $c = 6, n_0 = 3$ also works. We cannot take $c = 7$, because $7n \le 7n - 3$ is false for every n.

Worked example. Show $\frac{1}{2}n^2 - 3n = \Omega(n^2)$.

Try $c = \frac{1}{4}$. We need $\frac{1}{4}n^2 \le \frac{1}{2}n^2 - 3n$, i.e. $3n \le \frac{1}{4}n^2$, i.e. $12 \le n$. So $c = \frac{1}{4}, n_0 = 12$ works. ■

Big-Ω is what we use for **best-case statements** ("this algorithm always takes at least this long") and, far more importantly, for **problem lower bounds** ("*no* algorithm can solve this problem in less than this much time"). Searching an unsorted array of n elements is $\Omega(n)$ in this second, stronger sense: any correct algorithm must in the worst case look at every element, because an element it never looked at could have been the one it was searching for.

3.5 Big-Θ: The Asymptotically Tight Bound

Now we squeeze from both sides at once. This needs **two** constants.

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

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

In Equation (3.3), c_1 is the **lower multiplier** and c_2 is the **upper multiplier**; n_0 is the single threshold beyond which both inequalities hold. In plain words: f **is eventually sandwiched between two fixed multiples of** g — it grows at exactly the same rate as g, up to constant factors.

Figure 3.4 shows the sandwich: $f(n)$ runs in the corridor between $c_1 \cdot g(n)$ below and $c_2 \cdot g(n)$ above.

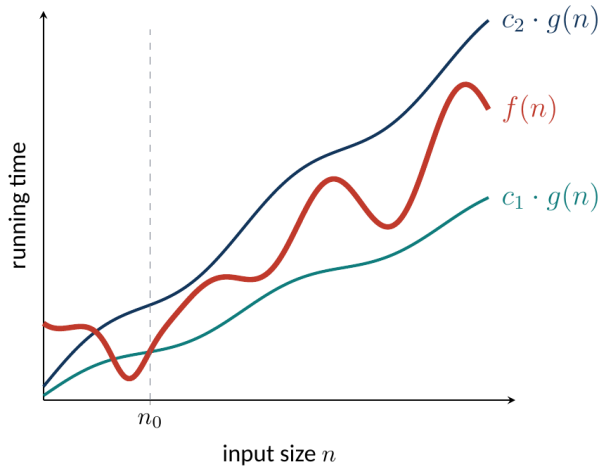


Figure 3.4 Big-Θ as a tight bound. Past n_0 the function $f(n)$ is trapped in the corridor between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$, so f and g grow at the same rate.

THEOREM. $f(n) = \Theta(g(n))$ **if and only if** $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Proof. ($\Rightarrow$) If Θ holds with constants c_1, c_2, n_0, then $f(n) \le c_2 g(n)$ for $n \ge n_0$ gives O with $c = c_2$, and $c_1 g(n) \le f(n)$ for $n \ge n_0$ gives Ω with $c = c_1$. ($\Leftarrow$) If $f = O(g)$ with constants c', n_1 and $f = \Omega(g)$ with constants c'', n_2, take $c_2 = c'$, $c_1 = c''$ and $n_0 = \max(n_1, n_2)$. Beyond that single threshold both inequalities hold at once. ■

This theorem is how Θ-proofs are almost always done in practice: prove the upper bound, prove the lower bound, combine.

Worked example, in full. Show $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Upper bound. For $n \ge 1$, $-3n \le 0$, hence

$$\frac{1}{2}n^2 - 3n \le \frac{1}{2}n^2, \qquad \text{so } c_2 = \frac{1}{2} \text{ works from } n \ge 1.$$

Lower bound. We want $c_1 n^2 \le \frac{1}{2}n^2 - 3n$. Rearranged,

$$\left(\frac{1}{2} - c_1\right) n^2 \geq 3n \Leftrightarrow \left(\frac{1}{2} - c_1\right) n \geq 3 \Leftrightarrow n \geq \frac{3}{\frac{1}{2} - c_1}.$$

Choose $c_1 = \frac{1}{4}$. Then the requirement becomes $n \geq 3/\frac{1}{4} = 12$.

Combine. With $c_1 = \frac{1}{4}$, $c_2 = \frac{1}{2}$ and $n_0 = 12$, both inequalities hold for all $n \geq 12$. Therefore $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. ■

THINK. O is very often used where Θ is meant. The statement $7n = O(n^5)$ is perfectly **true** — take $c = 1$, $n_0 = 1$ — and perfectly **useless**, because it throws away almost everything you knew. Always report the tightest bound you can prove. Saying "Insertion Sort is $O(n^2)$ in the worst case" is right; saying it is $\Theta(n^2)$ in the worst case is right *and* informative, because it also tells you the bound cannot be improved.

3.6 Little-o, Little-ω, and the Number Analogy

Big-O allows f and g to grow at the same rate: $2n^2 = O(n^2)$ is true. Sometimes we want to say f grows **strictly** slower. That is little-o.

DEFINITION (little-o). $f(n) = o(g(n))$ if for **every** positive constant c there exists a threshold n_0 (which may depend on c) such that

$$0 \leq f(n) < c \cdot g(n) \quad \text{for all } n \geq n_0.$$

(3.4)

Equivalently, when the limit exists, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Read the quantifier in Equation (3.4) carefully, because it is the whole difference. In Big-O we said "**there exists** a c". In little-o we say "**for every** c". No matter how small you shrink the multiplier — $c = 0.001$, $c = 10^{-9}$ — f eventually ducks underneath. That can only happen if f is genuinely, asymptotically negligible compared with g.

DEFINITION (little-ω). $f(n) = \omega(g(n))$ if for **every** positive constant c there exists n_0 such that

$$0 \leq c \cdot g(n) < f(n) \quad \text{for all } n \geq n_0.$$

Equivalently, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$. It is the strict version of Ω, and $f(n) = \omega(g(n))$ if and only if $g(n) = o(f(n))$.

Worked examples.

- $2n = o(n^2)$. Given any $c > 0$, we need $2n < cn^2$, i.e. $n > 2/c$. Take $n_0 = \lceil 2/c \rceil + 1$. Done — and note how n_0 depended on c, which is allowed.
- $2n^2 \neq o(n^2)$. Choose $c = 1$. We would need $2n^2 < n^2$, i.e. $n^2 < 0$, impossible. One bad c is enough to kill a little-o claim.
- $n^2 = \omega(n)$, because $n^2/n = n \to \infty$.
- $\lg n = o(n)$, because $\lg n/n \to 0$. Logarithms are strictly slower than any positive power of n.
- $n^{100} = o(2^n)$. Every polynomial is strictly slower than every exponential with base above 1.

3.6.1 The number analogy

The five notations line up with the five comparisons between ordinary numbers, and this is the single most useful mental picture in the chapter. **Table 3.2** makes the correspondence explicit.

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	$f \leq g$
$f(n) = \Omega(g(n))$	f grows at least as fast as g	$f \geq g$
$f(n) = \Theta(g(n))$	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	$f > g$

The analogy is a good servant and a bad master. It is reliable in one important respect and unreliable in another, and §3.7.6 explains where it breaks.

3.6.2 An abuse of notation we all commit

Strictly speaking, $O(g(n))$ is a **set of functions** — the set of all f for which suitable c and n_0 exist. The honest way to write the relation is

$$f(n) \in O(g(n)),$$

with the membership symbol $\in$. Everyone, this book included, writes $f(n) = O(g(n))$ with an equals sign instead. It is universally understood and universally sloppy. Two consequences of the sloppiness are worth flagging now:

1. **The equals sign is one-directional.** $2n = O(n^2)$ is true; "$O(n^2) = 2n$" is meaningless. Never read an asymptotic equation from right to left.
2. **Asymptotics inside a formula mean "some function".** When we write $T(n) = 2T(n/2) + \Theta(n)$, the $\Theta(n)$ stands for *some anonymous function* that happens to be $\Theta(n)$. When we write $n^2 + O(n)$, the $O(n)$ stands for some anonymous function bounded above by a multiple of n.

3.7 Properties of the Notations

The five notations obey a small algebra. Knowing it saves an enormous amount of work, because it lets you manipulate bounds without going back to the definition every time.

3.7.1 Transitivity

All five are transitive.

$$\begin{aligned} f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) &\implies f(n) = O(h(n)), \\ f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) &\implies f(n) = \Omega(h(n)), \\ f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) &\implies f(n) = \Theta(h(n)), \end{aligned}$$

and likewise for o and ω.

Proof for O. Suppose $f(n) \leq c_1 g(n)$ for $n \geq n_1$ and $g(n) \leq c_2 h(n)$ for $n \geq n_2$. For every $n \geq \max(n_1, n_2)$, both hold, so

$$f(n) \le c_1 g(n) \le c_1(c_2 h(n)) = (c_1 c_2)\, h(n).$$

Take $c = c_1 c_2$ and $n_0 = \max(n_1, n_2)$. ■

Transitivity is why chains of reasoning work: if the merge step is $O(n)$ and n is $O(n\lg n)$, then the merge step is $O(n\lg n)$.

3.7.2 Reflexivity

$f(n) = O(f(n))$, $f(n) = \Omega(f(n))$ and $f(n) = \Theta(f(n))$, always. Take $c = 1$ (or $c_1 = c_2 = 1$) and $n_0 = 1$.

Little-o and little-ω are **not** reflexive: $f(n) \ne o(f(n))$, since $f/f = 1 \ne 0$. This matches the number analogy — $x \le x$ is true, $x < x$ is false.

3.7.3 Symmetry and transpose symmetry

Symmetry holds only for Θ:

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n)).$$

Proof. From $c_1 g \le f \le c_2 g$ for $n \ge n_0$, divide through. The right inequality gives $g \ge f/c_2$, and the left gives $g \le f/c_1$. So $\frac{1}{c_2} f \le g \le \frac{1}{c_1} f$, which is the Θ definition with the roles swapped and constants $\frac{1}{c_2}, \frac{1}{c_1}$. ■

Transpose symmetry relates the upper notations to the lower ones:

$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n)), \qquad f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n)).$$

Proof of the first. $f(n) \le c\, g(n)$ for $n \ge n_0$ is literally the same statement as $g(n) \ge \frac{1}{c} f(n)$ for $n \ge n_0$, which is Ω with constant $1/c$. ■

In the number analogy: $f \le g$ says the same thing as $g \ge f$. Exactly right.

3.7.4 Sums, products, and constants

Let $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. Then:

- **Sum rule.** $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$.
- **Product rule.** $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.
- **Constant rule.** $k \cdot f_1(n) = O(g_1(n))$ for any constant $k > 0$.

Proof of the sum rule. Suppose $f_1 \le c_1 g_1$ for $n \ge n_1$ and $f_2 \le c_2 g_2$ for $n \ge n_2$. Write $g(n) = \max(g_1(n), g_2(n))$. For $n \ge \max(n_1, n_2)$,

$$f_1(n) + f_2(n) \le c_1 g_1(n) + c_2 g_2(n) \le c_1 g(n) + c_2 g(n) = (c_1 + c_2)\, g(n). \qquad ■$$

Proof of the product rule. For $n \ge \max(n_1, n_2)$, all quantities being non-negative,

$$f_1(n) f_2(n) \le (c_1 g_1(n))(c_2 g_2(n)) = (c_1 c_2)\, g_1(n) g_2(n). \qquad ■$$

The sum rule is where "drop the lower-order terms" comes from, and the constant rule is where "drop the constant factors" comes from. Let us see that spelled out.

3.7.5 Why we drop constants and lower-order terms — proved, not asserted

Constants. Suppose $f(n) = 3n^2$. Is $f(n) = \Theta(n^2)$? Take $c_1 = c_2 = 3$ and $n_0 = 1$; then $3n^2 \leq 3n^2 \leq 3n^2$ trivially. So yes. The constant simply becomes part of the witness. More generally, for any $k > 0$, $k \cdot g(n) = \Theta(g(n))$, because the definition already grants us an arbitrary multiplier.

The deeper reason is the one from §3.2: a constant factor is a fact about the machine, not about the algorithm. Change the compiler and the constant changes. Change the algorithm and the growth class changes.

Lower-order terms. Let $p(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0$ with $a_d > 0$. We claim $p(n) = \Theta(n^d)$.

Upper bound. For $n \geq 1$ every power n^i with $i \leq d$ satisfies $n^i \leq n^d$. Therefore

$$p(n) \leq (|a_d| + |a_{d-1}| + \cdots + |a_0|)n^d = An^d,$$

where A is the sum of the absolute values of the coefficients — a fixed number. So $c_2 = A$ works from $n_0 = 1$.

Lower bound. Write $p(n) = a_d n^d + R(n)$ where $R(n) = a_{d-1} n^{d-1} + \cdots + a_0$. By the same bound, $|R(n)| \leq Bn^{d-1}$ for $n \geq 1$, where $B = |a_{d-1}| + \cdots + |a_0|$. Hence

$$p(n) \geq a_d n^d - Bn^{d-1} = n^d \left(a_d - \frac{B}{n}\right).$$

As soon as $n \geq 2B/a_d$ we have $B/n \leq a_d/2$, so $p(n) \geq \frac{a_d}{2} n^d$. Take $c_1 = a_d/2$ and $n_0 = \max(1, \lceil 2B/a_d \rceil)$.

Conclusion. $p(n) = \Theta(n^d)$. **Every polynomial is Θ of its leading power.** ■

Applied to three typical examples:

- $50n\log n \to \Theta(n\log n)$: the 50 is absorbed by the constant.
- $7n - 3 \to \Theta(n)$: degree 1, leading coefficient 7.
- $8n^2\log n + 5n^2 + n \to \Theta(n^2\log n)$: here $n^2\log n$ outgrows n^2, which outgrows n, so the first term rules. (This is not a polynomial, but the same sum-rule argument applies: the largest term wins.)

3.7.6 Where the number analogy breaks: trichotomy fails

For real numbers, exactly one of $a < b$, $a = b$, $a > b$ holds. This is **trichotomy**, and it is what makes numbers comparable.

Asymptotic growth has **no** trichotomy. There are pairs of functions where none of $f = O(g)$, $f = \Theta(g)$, $f = \Omega(g)$ holds. The standard example is

$$f(n) = n, \qquad g(n) = n^{1+\sin n}.$$

The exponent $1 + \sin n$ oscillates between 0 and 2 forever. When $\sin n$ is near 1, g is near n^2 and towers over f; when $\sin n$ is near -1, g is near $n^0 = 1$ and f towers over g. Neither function is eventually above or eventually below the other, so neither O nor Ω holds in either direction.

COMMON MISCONCEPTION. "Every two functions can be compared asymptotically." They cannot. Functions that oscillate can be incomparable. In practice, the running times of real algorithms are built from polynomials, logarithms and exponentials, which are all mutually

comparable — so the analogy works day to day. But it is not a theorem, and the exception is exactly the kind of $f(n)$ that makes the Master Theorem's third case fail.

3.8 Standard Notations and Common Functions

Every running time in this book is assembled from a short list of building blocks. This section introduces each one, with the identities we will actually use.

3.8.1 Monotonicity

A function f is **monotonically increasing** if $m \le n$ implies $f(m) \le f(n)$; **monotonically decreasing** if $m \le n$ implies $f(m) \ge f(n)$. It is **strictly** increasing or decreasing if the inequalities are strict. Almost every running-time function we meet is monotonically increasing — more input, more work — and we use this constantly without saying so.

3.8.2 Floors and ceilings

For a real number x:

- $\lfloor x \rfloor$, the **floor**, is the greatest integer $\le x$. $\lfloor 3.7 \rfloor = 3$, $\lfloor 3 \rfloor = 3$, $\lfloor -3.2 \rfloor = -4$.
- $\lceil x \rceil$, the **ceiling**, is the least integer $\ge x$. $\lceil 3.2 \rceil = 4$, $\lceil 3 \rceil = 3$.

The identities we rely on:

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1, \qquad \lfloor x \rfloor + \lceil x \rceil = 2x \text{ when } x \in \mathbb{Z},$$

$$\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n \quad \text{for every integer } n.$$

That last identity is the one that makes Merge Sort's split work: whatever n is, odd or even, the two halves add back up to n exactly. Both floor and ceiling are monotonically increasing.

Recurrences from divide-and-conquer really look like $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$. We will almost always write $T(n) = 2T(n/2) + \Theta(n)$ instead, and §3.10.3 explains why that is harmless.

3.8.3 Modular arithmetic

For integers a and $n > 0$, $a \bmod n$ is the remainder: $a \bmod n = a - n\lfloor a/n \rfloor$, always in the range $0 \le a \bmod n < n$. Hash tables, cyclic buffers and the "wrap around" in many loops use it.

3.8.4 Polynomials

A **polynomial in n of degree d** is $p(n) = \sum_{i=0}^{d} a_i n^i$ with $a_d \ne 0$. We proved in §3.7.5 that if $a_d > 0$ then $p(n) = \Theta(n^d)$.

A function is **polynomially bounded** if $f(n) = O(n^k)$ for some constant k. Useful powers:

$$n^0 = 1, \quad n^1 = n, \quad n^{-1} = \frac{1}{n}, \quad n^{1/2} = \sqrt{n}, \quad n^a n^b = n^{a+b}, \quad (n^a)^b = n^{ab}.$$

3.8.5 Exponentials

For real $a > 0$ and integers m, n:

$$a^0 = 1, \qquad a^{-1} = \frac{1}{a}, \qquad (a^m)^n = a^{mn}, \qquad a^m a^n = a^{m+n}.$$

The single most important fact about exponentials in algorithm analysis:

THEOREM. For every constant $a > 1$ and every constant $d > 0$,

$$\lim_{n\to\infty} \frac{n^d}{a^n} = 0, \qquad \text{that is,} \qquad n^d = o(a^n).$$

Any exponential with base greater than 1 eventually beats any polynomial, however large its degree. So $n^{1000} = o(1.001^n)$, even though the polynomial is ahead for an absurdly long time.

This is why exponential-time algorithms are catastrophic and polynomial-time algorithms are (in principle) acceptable, and it is the reason the whole field draws its main line between the two.

Also useful, from the definition $e^x = \sum_{i=0}^{\infty} x^i / i!$:

$$e^x \geq 1 + x \quad \text{for all real } x, \qquad \lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x.$$

3.8.6 Logarithms — and the notation we will use everywhere

A logarithm answers the question "to what power must I raise the base to get this number?" If $b^y = x$ then $\log_b x = y$.

Three abbreviations are standard and we use all three:

$$\lg n = \log_2 n \quad \text{(binary)}, \qquad \ln n = \log_e n \quad \text{(natural)}, \qquad \lg^k n = (\lg n)^k, \qquad \lg\lg n = \lg(\lg n).$$

Note carefully that $\lg^2 n$ means $(\lg n)^2$, **not** $\lg(\lg n)$. This trips people up constantly; it will matter in §3.13 when a gap case produces $\Theta(n \lg^2 n)$.

Table 3.3 lists every identity we will use in this chapter, with a one-line reason for each.

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 **change-of-base** identity has a consequence that deserves its own line: since $\log_c b$ is a constant, all logarithms differ only by a constant factor, so

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

(3.5)

By Equation (3.5), inside O, Ω or Θ **the base of a logarithm does not matter** and is usually omitted. Outside them — in an exact count, or in an exponent like $n^{\log_b a}$ — it matters enormously.

The identity $a^{\log_b n} = n^{\log_b a}$ is the algebraic heart of the Master Theorem, so let us prove it slowly. Take $\log_b$ of the left-hand side:

$$\log_b(a^{\log_b n}) = (\log_b n)(\log_b a).$$

Now take $\log_b$ of the right-hand side:

$$\log_b(n^{\log_b a}) = (\log_b a)(\log_b n).$$

Multiplication commutes, so the two logarithms are equal; since $\log_b$ is a strictly increasing (hence one-to-one) function, the two original quantities are equal. ■

Finally, two growth facts:

$$\lg n = o(n^{\varepsilon}) \text{ for every constant } \varepsilon > 0, \qquad \lg(n!) = \Theta(n \lg n).$$

The first says logarithms are slower than *any* positive power of n, however tiny the power — which is exactly why $\lg n$ can never supply the polynomial separation the Master Theorem's Cases 1 and 3 demand.

3.8.7 Factorials and Stirling's approximation

$n! = 1 \cdot 2 \cdot 3 \cdots n$, with $0! = 1$ by convention. It counts the permutations of n objects. A crude bound is $n! \le n^n$, and a crude lower bound is $n! \ge 2^{n-1}$. **Stirling's approximation** is far sharper:

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

Three consequences of Equation (3.6) that we use: $n! = o(n^n)$, $n! = \omega(2^n)$, and, taking logarithms, $\lg(n!) = \Theta(n \lg n)$. That last equation is the reason no comparison sort can beat $n \lg n$ — a fact Chapter 5 proves.

3.8.8 The iterated logarithm

$$\lg^* n = \min\{\, i \ge 0 : \lg^{(i)} n \le 1 \,\},$$

where $\lg^{(i)}$ means "apply lg exactly i times". In words: **how many times must you take the logarithm before the number drops to 1 or below?** It grows so slowly as to be effectively constant: $\lg^* 2 = 1$, $\lg^* 4 = 2$, $\lg^* 16 = 3$, $\lg^* 65536 = 4$, and $\lg^* 2^{65536} = 5$ — and 2^{65536} has about 20,000 decimal digits, vastly more than the number of atoms in the observable universe.

3.8.9 Fibonacci numbers

$$F_0 = 0, \qquad F_1 = 1, \qquad F_i = F_{i-1} + F_{i-2} \text{ for } i \geq 2,$$

giving 0,1,1,2,3,5,8,13,21,34, ... With the golden ratio $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$ and its conjugate $\hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.61803$,

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}, \qquad \text{and since } |\hat{\phi}| < 1, \qquad F_i = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor.$$

So the Fibonacci numbers grow **exponentially**, at rate ϕ^n. This is exactly why the naive recursive Fibonacci program of Chapter 1 is unusable, and why Chapter 7 rewrites it with memoization.

Listing 3.3 makes three of these facts visible.

Listing 3.3 The iterated logarithm, Stirling's approximation and the Fibonacci closed form, computed and compared.

```
// functions.cpp - The standard functions of Chapter 3 seen numerically:
// the iterated logarithm, Stirling's approximation to n!, and the closed
// form for the Fibonacci numbers.
// Compile: g++ -std=c++17 -O2 -o functions functions.cpp
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

// lg*(n): how many times must we take lg before the value drops to <= 1?
int lgStar(double n) {
    int count = 0;
    while (n > 1.0) { n = log2(n); ++count; }
    return count;
}

int main() {
    cout << "-- The iterated logarithm lg*(n) --\n";
    double vals[] = {1, 2, 4, 16, 65536, pow(2.0, 65536.0)};
    const char* names[] = {"1", "2", "4", "16", "65536", "2^65536"};
    for (int i = 0; i < 6; ++i) {
        if (isinf(vals[i])) { cout << "  n = " << setw(10) << names[i] << "  lg*(n) = 5 (by
definition, one more than lg*(65536))\n"; continue; }
        cout << "  n = " << setw(10) << names[i] << "  lg*(n) = " << lgStar(vals[i]) <<
"\n";
    }

    cout << "\n-- Stirling: n! vs sqrt(2 pi n) (n/e)^n --\n";
    cout << fixed;
    double fact = 1.0;
    for (int n = 1; n <= 20; ++n) {
        fact *= n;
        if (n == 1 || n == 5 || n == 10 || n == 15 || n == 20) {
            double st = sqrt(2 * M_PI * n) * pow(n / M_E, n);
            cout << "  n = " << setw(3) << n
                 << "  n! = " << scientific << setprecision(6) << setw(14) << fact
                 << "  Stirling = " << setw(14) << st
                 << "  ratio = " << fixed << setprecision(6) << fact / st << "\n";
        }
    }
```

Listing 3.3 (Continued) *The iterated logarithm, Stirling's approximation and the Fibonacci closed form, computed and compared.*

```
    cout << "\n-- Fibonacci F(n) vs phi^n / sqrt(5), phi = (1+sqrt 5)/2 --\n";
    double phi = (1.0 + sqrt(5.0)) / 2.0;
    long long a = 0, b = 1;
    for (int n = 1; n <= 40; ++n) {
        long long t = a + b; a = b; b = t;   // after the loop body, a = F(n)
        if (n == 1 || n == 5 || n == 10 || n == 20 || n == 40)
            cout << "  n = " << setw(3) << n << "  F(n) = " << setw(12) << a
                 << "  phi^n/sqrt5 = " << setprecision(3) << pow(phi, n) / sqrt(5.0) <<
"\n";
    }
    cout << "\n  phi = " << setprecision(9) << phi
         << "  so F(n) grows like " << setprecision(3) << phi << "^n\n\n";
    return 0;
}
```

The output:

```
-- The iterated logarithm lg*(n) --
  n =          1  lg*(n) = 0
  n =          2  lg*(n) = 1
  n =          4  lg*(n) = 2
  n =         16  lg*(n) = 3
  n =      65536  lg*(n) = 4
  n =    2^65536  lg*(n) = 5 (by definition, one more than lg*(65536))

-- Stirling: n! vs sqrt(2 pi n) (n/e)^n --
  n =   1  n! =   1.000000e+00  Stirling =   9.221370e-01  ratio = 1.084438
  n =   5  n! =   1.200000e+02  Stirling =   1.180192e+02  ratio = 1.016784
  n =  10  n! =   3.628800e+06  Stirling =   3.598696e+06  ratio = 1.008365
  n =  15  n! =   1.307674e+12  Stirling =   1.300431e+12  ratio = 1.005570
  n =  20  n! =   2.432902e+18  Stirling =   2.422787e+18  ratio = 1.004175
```

Two things to notice. Going from $n = 65536$ to $n = 2^{65536}$ — an unimaginable leap — raises $\lg^* n$ by exactly one. And Stirling's ratio marches steadily toward 1: it is 1.0844 at $n = 1$ and 1.0042 at $n = 20$, consistent with the $1 + \Theta(1/n)$ correction term.

The Fibonacci half of the same run prints:

```
-- Fibonacci F(n) vs phi^n / sqrt(5), phi = (1+sqrt 5)/2 --
  n =   1  F(n) =            1  phi^n/sqrt5 = 0.724
  n =   5  F(n) =            5  phi^n/sqrt5 = 4.960
  n =  10  F(n) =           55  phi^n/sqrt5 = 55.004
  n =  20  F(n) =         6765  phi^n/sqrt5 = 6765.000
  n =  40  F(n) =    102334155  phi^n/sqrt5 = 102334155.000

  phi = 1.618033989  so F(n) grows like 1.618^n
```

By $n = 20$ the closed form is already exact to the digits printed — the $\hat{\phi}^i$ correction has shrunk below 10^{-4}. **Table 3.4** gathers the whole family in growth 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

3.9 Summations: the Arithmetic of Loops

A loop that runs n times and does t_i work on iteration i costs $\sum_{i=1}^{n} t_i$. Nested loops give nested sums. Recursion trees, later in this chapter, give sums over levels. Summations are therefore the working arithmetic of algorithm analysis, and this section derives the four we need.

First, the notation itself. The symbol

$$\sum_{i=1}^{n} a_i$$

is read "the sum, as i runs from 1 to n, of a_i", and it simply means $a_1 + a_2 + \cdots + a_n$. The letter i is the **index**; 1 is the **lower limit**; n is the **upper limit**; a_i is the **summand**. If the upper limit is below the lower limit the sum is **empty** and equals 0 by convention. Two rules follow directly from ordinary arithmetic and get used constantly:

$$\sum_{i=1}^{n} (c \cdot a_i) = c \sum_{i=1}^{n} a_i \qquad \text{(linearity: constants come out)},$$

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i \qquad \text{(linearity: sums split)}.$$

3.9.1 The arithmetic series

DEFINITION (arithmetic series).

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

(3.7)

Figure 3.5 states Equation (3.7) in a slightly different form; note that it starts the index at $i = 0$, which changes nothing because the term $i = 0$ contributes zero, and writes the closed form as $(n^2 + n)/2$, which is the same thing as $n(n + 1)/2$.

$$\sum_{i=0}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n^2 + n}{2}$$

Figure 3.5 The arithmetic series and its closed form, $\sum_{i=0}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n^2+n}{2}$.

Gauss's trick, the derivation. The story is that Gauss found this at about nine years old when a teacher set the class to add 1 through 100. Write the sum forward, then write it again backward underneath:

$$\begin{array}{llllll} S = 1 & +\,2 & +\,3 & +\cdots & +\,(n-1) & +\,n \\ S = n & +(n-1) & +(n-2) & +\cdots & +\,2 & +\,1 \end{array}$$

Now add the two lines **column by column**. The first column gives $1 + n$. The second gives $2 + (n - 1) = n + 1$. The third gives $3 + (n - 2) = n + 1$. Every column gives the same total, $n + 1$, because as one addend goes up by one the other goes down by one. There are n columns. So

$$2S = n(n + 1), \qquad \text{hence} \qquad S = \frac{n(n + 1)}{2}. \qquad \blacksquare$$

Figure 3.6 is the picture of the same argument: the staircase of bars of heights $1, 2, \ldots, n$ is folded into a solid rectangle $n/2$ wide and $n + 1$ tall, so its area is $\frac{n}{2}(n + 1)$.

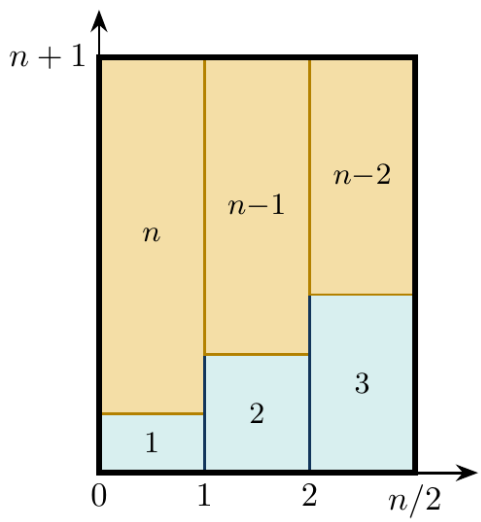


Figure 3.6 Gauss's pairing made geometric. Pairing the first bar with the last, the second with the second-last, and so on, repacks the staircase into a rectangle of width $n/2$ and height $n + 1$, of area $n(n + 1)/2$.

A second picture gives the same answer a different way, and is worth having because it generalizes. In **Figure 3.7** the bars of heights 1,2,3, … , n stand side by side under the straight line through the origin of slope 1. The area under that line from 0 to n is the triangle $\frac{1}{2}n \cdot n = \frac{n^2}{2}$. The bars overhang the line by exactly half a unit square each, n of them, adding $\frac{n}{2}$. Total: $\frac{n^2}{2} + \frac{n}{2} = \frac{n(n+1)}{2}$, as before.

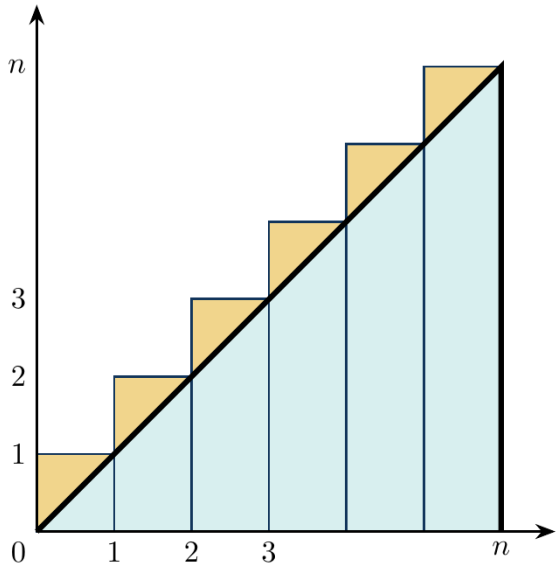


Figure 3.7 The arithmetic series as area. The bars of heights 1,2, … , n sit astride the line of slope 1; their total area $n(n+1)/2$ is the triangle $n^2/2$ plus the n half-squares that overhang it.

Why $\Theta(n^2)$? Because $\frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$ is a polynomial of degree 2 with positive leading coefficient, and §3.7.5 proved every such polynomial is Θ of its leading power. Concretely, $\frac{n^2}{2} \le \frac{n(n+1)}{2} \le n^2$ for $n \ge 1$, so $c_1 = 1/2$, $c_2 = 1$, $n_0 = 1$.

Where it shows up. Insertion Sort's worst case. The inner `while` loop for index j can shift as many as $j - 1$ elements, so the total number of shifts is at most

$$\sum_{j=2}^{n} (j-1) = 1 + 2 + \cdots + (n-1) = \frac{(n-1)n}{2} = \Theta(n^2).$$

That is the entire worst-case analysis of Insertion Sort, in one line, now that we have the formula.

3.9.2 The same result by induction

The closed form can also be proved by **mathematical induction**, and it is worth doing once because induction is the engine of the substitution method in §3.11. The recipe has two parts:

1. **Basis.** Prove the claim $P(n_0)$ for the smallest case.
2. **Inductive step.** Assume $P(k)$ for all $n_0 \le k \le n-1$ (this is *strong* induction), and prove $P(n)$.

Claim. $S(n) = \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ for all $n \ge 1$.

Basis ($n = 1$). The left side is $S(1) = 0 + 1 = 1$. The right side is $\frac{1 \cdot 2}{2} = 1$. They agree. ✓

Inductive step. Assume $S(k) = \frac{k(k+1)}{2}$ for all $k \le n-1$. Then

$$S(n) = \sum_{i=0}^{n} i = \left(\sum_{i=0}^{n-1} i\right) + n = S(n-1) + n$$

$$= \frac{(n-1)((n-1)+1)}{2} + n \qquad \text{(inductive hypothesis at } k = n-1\text{)}$$

$$= \frac{(n-1)n}{2} + n = \frac{n^2 - n}{2} + \frac{2n}{2} = \frac{n^2 - n + 2n}{2} = \frac{n^2 + n}{2} = \frac{n(n+1)}{2}. \checkmark$$

By induction the formula holds for every $n \ge 1$. ■ **Figure 3.8** shows exactly this chain of equalities written out.

$$S(k) = \sum_{i=0}^{k} i = \frac{k(k+1)}{2} \quad \text{for } 1 \le k \le n-1,$$

$$S(n) = \sum_{i=0}^{n} i = \sum_{i=0}^{n-1} i + n = S(n-1) + n$$

$$= \frac{(n-1)(n-1+1)}{2} + n = \frac{n^2 - n + 2n}{2}$$

$$= \frac{n(n+1)}{2}.$$

Figure 3.8 The inductive step written out: assume $S(k) = k(k+1)/2$ for $1 \le k \le n-1$, then unfold $S(n) = S(n-1) + n$ and simplify to $n(n+1)/2$.

3.9.3 The geometric series

DEFINITION (geometric series). For a real $a \ne 1$,

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

(3.8)

Figure 3.9 states it in the same form.

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{1 - a^{n+1}}{1 - a} \qquad (a \ne 1)$$

Figure 3.9 The finite geometric series, $\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{1-a^{n+1}}{1-a}$.

Derivation of Equation (3.8)**, by the same trick as Gauss but with multiplication instead of reversal.** Let $S = 1 + a + a^2 + \cdots + a^n$. Multiply every term by a:

$$aS = a + a^2 + a^3 + \cdots + a^{n+1}.$$

Subtract the second line from the first. Everything in the middle cancels — a cancels a, a^2 cancels a^2, and so on up to a^n — leaving only the first term of S and the last term of aS:

$$S - aS = 1 - a^{n+1}, \qquad \text{so} \qquad S(1-a) = 1 - a^{n+1}, \qquad S = \frac{1-a^{n+1}}{1-a}. \qquad \blacksquare$$

(The step divides by $1-a$, which is why $a \neq 1$ is required. When $a = 1$ every term is 1 and the sum is simply $n+1$.)

Three regimes, and they are the whole of recursion-tree analysis:

- **If $a < 1$ (convergent).** As $n \to \infty$, $a^{n+1} \to 0$, so the sum converges to $\frac{1}{1-a}$, a **constant**. The series is $\Theta(1)$, and the **first term dominates** — the whole infinite tail is worth no more than a constant multiple of the first term. Example: $1 + 1/2 + 1/4 + \cdots = 2$.
- **If $a > 1$ (divergent).** Then a^{n+1} dwarfs the 1, and $\frac{a^{n+1}-1}{a-1} = \Theta(a^n)$. The **last term dominates**. Example: $1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1 = \Theta(2^n)$.
- **If $a = 1$.** The sum is $n + 1 = \Theta(n)$. **Every term contributes equally.**

> **KEY INSIGHT.** A geometric sum is always dominated by its largest term, up to a constant factor. If the ratio is below 1, the largest term is the first; if above 1, the last; if exactly 1, there is no single largest and you multiply one term by the number of terms. Recognizing which of the three you are in is *the entire skill* of recursion-tree analysis — and, as §3.13 will show, it is also exactly the three cases of the Master Theorem.

3.9.4 The harmonic series

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

In Equation (3.9), $\gamma \approx 0.5772156649$ is the Euler–Mascheroni constant. The harmonic series diverges — it grows without bound — but only logarithmically slowly. It appears in the average-case analysis of Quicksort (Chapter 5) and in randomized algorithms generally.

A one-line proof that $H_n = \Omega(\lg n)$. Group the terms in blocks of doubling length:

$$1 + \underbrace{\frac{1}{2}}_{\geq 1/2} + \underbrace{\left(\frac{1}{3} + \frac{1}{4}\right)}_{\geq 2 \cdot \frac{1}{4} = 1/2} + \underbrace{\left(\frac{1}{5} + \cdots + \frac{1}{8}\right)}_{\geq 4 \cdot \frac{1}{8} = 1/2} + \cdots$$

Each block of 2^k terms is at least $2^k \cdot 2^{-(k+1)} = 1/2$, and there are about $\lg n$ blocks, so $H_n \geq 1/2 \lg n$. $\blacksquare$

3.9.5 Telescoping sums

A sum **telescopes** when consecutive terms cancel:

$$\sum_{i=1}^{n} (a_i - a_{i-1}) = a_n - a_0.$$

Everything in the middle appears once with a plus and once with a minus. The classic example:

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \sum_{i=1}^{n} \left(\frac{1}{i} - \frac{1}{i+1}\right) = 1 - \frac{1}{n+1},$$

because $\frac{1}{i} - \frac{1}{i+1} = \frac{(i+1)-i}{i(i+1)} = \frac{1}{i(i+1)}$. Telescoping is also the cleanest way to solve simple recurrences: if $T(n) - T(n-1) = f(n)$, then summing both sides from 1 to n gives $T(n) - T(0) = \sum_{i=1}^{n} f(i)$.

3.9.6 Bounding a sum by an integral

When no closed form exists, compare the sum with an integral. If f is **monotonically increasing**, then each term $f(i)$ is at most the area of f over $[i, i+1]$ and at least the area over $[i-1, i]$, giving

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

If f is monotonically **decreasing**, the inequalities of Equation (3.10) swap. **Figure 3.7** is precisely this picture for $f(x) = x$: the bars are the sum, the line is the integrand.

Example 1 — the arithmetic series again. With $f(x) = x$ increasing,

$$\sum_{i=1}^{n} i \;\le\; \int_{1}^{n+1} x\,dx = \left[\frac{x^2}{2}\right]_{1}^{n+1} = \frac{(n+1)^2 - 1}{2} = \frac{n^2 + 2n}{2} = O(n^2).$$

Example 2 — the harmonic series. With $f(x) = 1/x$ decreasing,

$$\sum_{i=1}^{n} \frac{1}{i} \;\le\; 1 + \int_{1}^{n} \frac{dx}{x} = 1 + \ln n = O(\lg n),$$

which together with §3.9.4 gives $H_n = \Theta(\lg n)$.

Example 3 — $\lg(n!)$**.** Since $\lg(n!) = \sum_{i=1}^{n} \lg i$ and lg is increasing,

$$\sum_{i=1}^{n} \lg i \;\le\; \int_{1}^{n+1} \lg x\,dx = \Theta(n \lg n), \qquad \sum_{i=1}^{n} \lg i \;\ge\; \int_{1}^{n} \lg x\,dx = \Theta(n \lg n),$$

so $\lg(n!) = \Theta(n \lg n)$, confirming §3.8.7 without Stirling.

Table 3.5 is the 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 $\lvert a \rvert < 1$	$\frac{1}{1-a}$	$\Theta(1)$
$\sum_{i=1}^{n} \frac{1}{i}$	$\ln n + \gamma + O(1/n)$	$\Theta(\lg n)$
$\sum_{i=1}^{n} \lg i$	$\lg(n!)$	$\Theta(n \lg n)$
$\sum_{i=1}^{n} (a_i - a_{i-1})$	$a_n - a_0$	telescoping

Listing 3.4 checks four of these rows numerically, against brute-force addition.

Listing 3.4 Brute-force addition checked against every closed form derived in §3.9.

```
// sums.cpp - Checking the closed forms for the four summations we will use
// over and over: arithmetic, geometric, harmonic, telescoping.
// Compile: g++ -std=c++17 -O2 -o sums sums.cpp
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main() {
    cout << fixed << setprecision(6);

    cout << "-- Arithmetic: sum of i for i = 1..n, against n(n+1)/2 --\n";
    for (long long n : {5LL, 10LL, 100LL, 1000LL}) {
        long long brute = 0;
        for (long long i = 1; i <= n; ++i) brute += i;
        long long closed = n * (n + 1) / 2;
        cout << "  n = " << setw(5) << n << "  brute = " << setw(8) << brute
             << "  closed = " << setw(8) << closed
             << (brute == closed ? "  MATCH\n" : "  DIFFER\n");
    }

    cout << "\n-- Geometric with ratio a = 1/2 (convergent): sum a^i, i = 0..n --\n";
    double s = 0.0;
    for (int n = 0; n <= 30; ++n) {
        s += pow(0.5, n);
        if (n == 0 || n == 3 || n == 10 || n == 30)
            cout << "  n = " << setw(3) << n << "  sum = " << s
                 << "   limit 1/(1-a) = 2.000000\n";
    }

    cout << "\n-- Geometric with ratio a = 2 (divergent): sum 2^i, i = 0..n --\n";
    for (int n : {3, 5, 10, 20}) {
        double t = 0.0;
        for (int i = 0; i <= n; ++i) t += pow(2.0, i);
        cout << "  n = " << setw(3) << n << "  sum = " << setw(12) << t
             << "   2^(n+1)-1 = " << pow(2.0, n + 1) - 1 << "\n";
    }
```

Listing 3.4 (Continued) *Brute-force addition checked against every closed form derived in §3.9.*

```
    cout << "\n-- Harmonic: H(n) = sum 1/i, against ln n + 0.5772156649 --\n";
    double h = 0.0;
    int next = 1;
    for (int i = 1; i <= 1000000; ++i) {
        h += 1.0 / i;
        if (i == next) {
            cout << "  n = " << setw(8) << i << "  H(n) = " << h
                 << "   ln n + gamma = " << log((double)i) + 0.5772156649 << "\n";
            next *= 100;
        }
    }

    cout << "\n-- Telescoping: sum 1/(i(i+1)) for i = 1..n equals 1 - 1/(n+1) --\n";
    for (int n : {1, 5, 50, 5000}) {
        double t = 0.0;
        for (int i = 1; i <= n; ++i) t += 1.0 / ((double)i * (i + 1));
        cout << "  n = " << setw(6) << n << "  sum = " << t
             << "   1 - 1/(n+1) = " << 1.0 - 1.0 / (n + 1) << "\n";
    }
    return 0;
}
```

Its output:

```
-- Arithmetic: sum of i for i = 1..n, against n(n+1)/2 --
  n =     5  brute =       15  closed =       15  MATCH
  n =    10  brute =       55  closed =       55  MATCH
  n =   100  brute =     5050  closed =     5050  MATCH
  n =  1000  brute =   500500  closed =   500500  MATCH

-- Geometric with ratio a = 1/2 (convergent): sum a^i, i = 0..n --
  n =   0  sum = 1.000000   limit 1/(1-a) = 2.000000
  n =   3  sum = 1.875000   limit 1/(1-a) = 2.000000
  n =  10  sum = 1.999023   limit 1/(1-a) = 2.000000
  n =  30  sum = 2.000000   limit 1/(1-a) = 2.000000

-- Geometric with ratio a = 2 (divergent): sum 2^i, i = 0..n --
  n =   3  sum =    15.000000   2^(n+1)-1 = 15.000000
  n =   5  sum =    63.000000   2^(n+1)-1 = 63.000000
  n =  10  sum =  2047.000000   2^(n+1)-1 = 2047.000000
  n =  20  sum = 2097151.000000   2^(n+1)-1 = 2097151.000000

-- Harmonic: H(n) = sum 1/i, against ln n + 0.5772156649 --
  n =        1  H(n) = 1.000000   ln n + gamma = 0.577216
  n =      100  H(n) = 5.187378   ln n + gamma = 5.182386
  n =    10000  H(n) = 9.787606   ln n + gamma = 9.787556
  n =  1000000  H(n) = 14.392727   ln n + gamma = 14.392726

-- Telescoping: sum 1/(i(i+1)) for i = 1..n equals 1 - 1/(n+1) --
  n =      1  sum = 0.500000   1 - 1/(n+1) = 0.500000
  n =      5  sum = 0.833333   1 - 1/(n+1) = 0.833333
  n =     50  sum = 0.980392   1 - 1/(n+1) = 0.980392
  n =   5000  sum = 0.999800   1 - 1/(n+1) = 0.999800
```

Note Gauss's famous answer sitting in the third line: 5050. Note too that the convergent geometric sum is still below 2 after thirty terms and always will be, while the divergent one has passed two million after twenty — that is the difference between "root dominates" and "leaves dominate", which we are about to meet again under different names. And after a *million* terms the harmonic sum has only reached 14.39, agreeing with $\ln n + \gamma$ to six digits.

3.10 Recurrences: Equations That Describe Recursion

3.10.1 What a recurrence is

> **DEFINITION.** A **recurrence** is an equation or inequality that describes a function in terms of its value on smaller inputs, together with one or more **boundary conditions** (also called base cases) that give the value outright for the smallest inputs.

The running time of any algorithm that calls itself is naturally described by a recurrence, because the algorithm itself is described that way: "to sort n things, sort two groups of $n/2$ things, then spend $\Theta(n)$ merging."

Here is that sentence turned into an equation. **Figure 3.10** states it as a two-case definition.

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

Figure 3.10 The Merge Sort recurrence as a two-case definition: $T(n) = \Theta(1)$ when $n = 1$, and $T(n) = 2T(n/2) + \Theta(n)$ when $n > 1$.

Read it aloud: *when the array has one element, sorting costs a constant amount; otherwise it costs two recursive sorts of half-size arrays, plus linear work.* The base case is not decoration. Without it the recurrence defines nothing at all, because the unfolding never stops.

3.10.2 The general divide-and-conquer form

Most recurrences in this book have the shape

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

and it is worth naming each piece of Equation (3.11) in words before we go further.

- $a \geq 1$ is **how many subproblems** the algorithm creates. It is a count, so it is at least 1.
- $b > 1$ is **the factor by which the input shrinks**. Each subproblem has size n/b. It must exceed 1, or the subproblems never get smaller and the recursion never ends.
- $f(n)$ is **the cost of everything that is not a recursive call**: splitting the input, and combining the answers. It must be asymptotically positive — positive for all sufficiently large n.

Note that a and b are **independent**. You might split into two halves and recurse on both ($a = 2, b = 2$, Merge Sort); split into two halves and recurse on only one ($a = 1, b = 2$, binary search); or cut an $n \times n$ matrix into four blocks of side $n/2$ and make seven recursive calls on such blocks ($a = 7, b = 2$, Strassen's matrix multiplication, §3.14.10).

3.10.3 Three technicalities we are allowed to ignore

Floors and ceilings. The honest recurrence for Merge Sort is $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$. We write $2T(n/2)$. It can be proved that for the recurrences in this chapter the floors and ceilings change the answer by at most a constant factor, which Θ absorbs. So we drop them, and reason as though n were always an exact power of b.

Boundary conditions. We usually omit the base case and write only $T(n) = 2T(n/2) + \Theta(n)$. The justification is that $T(n) = \Theta(1)$ for all n below some constant, and changing a constant number of constant-sized values cannot change the asymptotic answer. When we need a concrete number for a worked computation we simply set $T(1) = 1$.

Equalities and inequalities. A recurrence stated with $\leq$ gives an O-bound; stated with $\geq$ it gives an Ω-bound; stated with $=$ it gives Θ. Keep track of which you have.

3.10.4 The three methods

Figure 3.11 lists the three techniques this chapter teaches, in the order we will teach them.

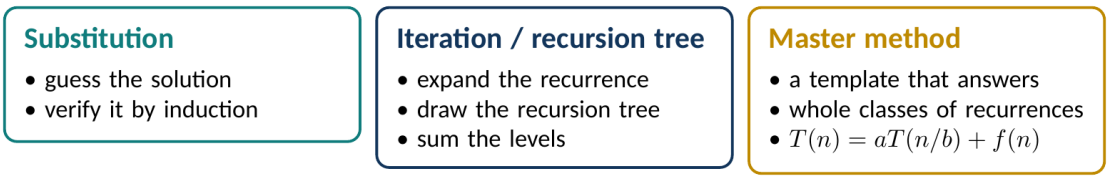


Figure 3.11 The three methods for solving recurrences: substitution (guess, then verify by induction), iteration or recursion tree (expand the recurrence and draw it), and the master method (a template that answers whole classes of recurrences at once).

They divide the labor neatly:

- the **recursion tree** *finds* an answer, informally but quickly;
- **substitution** *proves* the answer, rigorously;

- the **master method** *skips* both, whenever the recurrence fits its template.

In practice you try the master method first. If the recurrence does not fit — or falls in one of its gaps — you draw a tree to get a guess, then use substitution to make it a proof.

3.11 Method 1: The Substitution Method

The substitution method has exactly two steps.

1. **Guess** the form of the solution.
2. **Verify** the guess by mathematical induction, and in doing so, find the constants that make it work.

The name is slightly misleading: what you substitute is the inductive hypothesis into the recurrence. It is the only one of the three methods that produces a genuine proof, so it is the one to reach for when a bound must be airtight.

3.11.1 A complete example: $T(n) = 4T(n/2) + n$ is $O(n^3)$

The guess. $T(n) \leq c\,n^3$ for some constant $c > 0$ and all $n \geq n_0$.

The inductive hypothesis. Assume the bound holds for all smaller inputs: $T(k) \leq c\,k^3$ for every $k < n$. In particular it holds for $k = n/2$.

The inductive step, one line at a time.

$$T(n) = 4\,T(n/2) + n$$

$$\leq 4 \cdot c\left(\frac{n}{2}\right)^3 + n \qquad \text{(substituting the hypothesis at } k = n/2\text{)}$$

$$= 4 \cdot c \cdot \frac{n^3}{8} + n \qquad \text{because } (n/2)^3 = n^3/8$$

$$= \frac{c\,n^3}{2} + n$$

$$= c\,n^3 - \left(\frac{c\,n^3}{2} - n\right) \qquad \text{(rewriting, to expose the slack)}$$

$$\leq c\,n^3 \qquad \text{provided } \frac{c\,n^3}{2} - n \geq 0.$$

Finding the constant. The condition $\frac{cn^3}{2} \geq n$ rearranges to $c \geq \frac{2}{n^2}$. Since $n \geq 1$ gives $\frac{2}{n^2} \leq 2$, the single choice $c \geq 2$ works for every $n \geq 1$.

The base case. We must also check the induction starts. With $T(1) = 1$ we need $1 \leq c \cdot 1^3 = c$, true for $c \geq 2$. ✓

Conclusion. $T(n) \leq 2n^3$ for all $n \geq 1$, so $T(n) = O(n^3)$. ■

The rewriting step — turning $\frac{cn^3}{2} + n$ into $cn^3 - \left(\frac{cn^3}{2} - n\right)$ — is the move to learn. It is not algebraic cleverness; it is bookkeeping. It puts the answer you want on the left and everything you must dispose of in a single visible bracket.

3.11.2 The failed attempt: is it $O(n^2)$?

The bound $O(n^3)$ is correct but loose. The recursion tree in §3.12 will show the true answer is $\Theta(n^2)$. So let us try to prove $O(n^2)$ directly — and watch it fail, because the failure is instructive.

The guess. $T(n) \leq c\,n^2$.

$$T(n) = 4T(n/2) + n \leq 4c\left(\frac{n}{2}\right)^2 + n = 4c \cdot \frac{n^2}{4} + n = c\,n^2 + n.$$

We need $cn^2 + n \leq cn^2$, which requires $n \leq 0$. **Impossible**, for every choice of $c > 0$. **Figure 3.12** records this dead end.

$$\begin{aligned}
&\text{Try to show } T(n) = O(n^2).\\
&\text{Assume } T(k) \leq ck^2 \text{ for } k < n.\\
T(n) &= 4T(n/2) + n\\
&\leq 4c(n/2)^2 + n\\
&= cn^2 + n\\
&\not\leq cn^2 \quad \text{for } \textit{no} \text{ choice of } c > 0.
\end{aligned}$$

Figure 3.12 The substitution method failing. Assuming $T(k) \leq ck^2$ yields $T(n) \leq cn^2 + n$, and $cn^2 + n \leq cn^2$ holds for **no** choice of $c > 0$.

> **COMMON TRAP.** Arriving at $T(n) \leq cn^2 + n$ and declaring "$cn^2 + n$ is $O(n^2)$, so we are done" is **wrong**, and it is the single most common error in substitution proofs. The induction must reproduce the hypothesis *exactly*, with the *same* constant c. Any leftover positive term, however small its order, breaks the induction: apply the sloppy step n times and the leftovers accumulate into something that is not $O(n^2)$ at all.

3.11.3 The fix: strengthen the hypothesis

The cure is counter-intuitive: **prove something stronger**. Subtract a lower-order term from the guess, so the induction has slack to absorb the leftover.

The strengthened guess. $T(n) \leq c_1 n^2 - c_2 n$ for constants $c_1, c_2 > 0$.

The inductive step, one line at a time.

$$T(n) = 4\,T(n/2) + n$$

$$\leq 4\left(c_1\left(\frac{n}{2}\right)^2 - c_2\left(\frac{n}{2}\right)\right) + n \qquad \left(\text{hypothesis at } k = n/2\right)$$

$$=4\left(\frac{c_1n^2}{4}-\frac{c_2n}{2}\right)+n$$

$$=c_1n^2-2c_2n+n$$

$$=c_1n^2-c_2n-(c_2n-n) \qquad \text{(exposing the slack again)}$$

$$\leq c_1n^2-c_2n \qquad \text{provided } c_2n-n\geq 0, \text{ i.e. } c_2\geq 1.$$

The induction now closes. **Figure 3.13** writes out exactly this chain.

$$\begin{aligned}
&\text{Assume } T(k)\leq c_1k^2-c_2k \text{ for } k<n.\\
T(n) &= 4T(n/2)+n\\
&\leq 4\big(c_1(n/2)^2-c_2(n/2)\big)+n\\
&= c_1n^2-2c_2n+n\\
&= c_1n^2-c_2n-(c_2n-n)\\
&\leq c_1n^2-c_2n \quad \text{if } c_2\geq 1.
\end{aligned}$$

Figure 3.13 The strengthened hypothesis $T(k)\leq c_1k^2-c_2k$ closes the induction: the extra $-c_2n$ absorbs the leftover $+n$ whenever $c_2\geq 1$.

Why does proving *more* make it *easier*? Because the inductive hypothesis is what you are allowed to assume, as well as what you must prove. A stronger hypothesis gives you a stronger assumption at $k=n/2$, and here the extra $-c_2(n/2)$, multiplied by the four subproblems, becomes $-2c_2n$ — twice as much negative slack as you must produce. The surplus $-c_2n$ pays for the $+n$.

Finally, since $c_1n^2-c_2n\leq c_1n^2$, the strengthened bound implies $T(n)=O(n^2)$, which is what we wanted all along.

> **KEY INSIGHT.** When induction leaves you with (desired bound) + (positive leftover), subtract a term of the leftover's own order from the hypothesis and try again. Do **not** add a term — that makes the hypothesis weaker and the leftover worse.

3.11.4 Two more substitution techniques

Beware of the base case. Sometimes the inductive step works but the base case does not. For $T(n)=2T(\lfloor n/2\rfloor)+n$ with $T(1)=1$, the guess $T(n)\leq cn\lg n$ fails at $n=1$, because $c\cdot 1\cdot \lg 1=0<1$. The repair is to move the base case: asymptotic bounds need only hold for $n\geq n_0$, so prove the base at $n=2$ and $n=3$ instead. We have $T(2)=4$ and $T(3)=5$; the requirements are $4\leq 2c\lg 2=2c$ and $5\leq 3c\lg 3\approx 4.75c$, both satisfied by $c\geq 2$. The inductive step for $n\geq 4$ only ever refers back to $\lfloor n/2\rfloor\geq 2$, so $n=1$ is never needed.

Changing variables. The recurrence $T(n)=2T(\sqrt{n})+\lg n$ does not look like anything we can handle. Put $m=\lg n$, so $n=2^m$ and $\sqrt{n}=2^{m/2}$:

$$T(2^m) = 2\,T(2^{m/2}) + m.$$

Now define $S(m) = T(2^m)$, giving $S(m) = 2S(m/2) + m$ — the Merge Sort recurrence, whose answer we know is $\Theta(m\lg m)$. Changing back, $m = \lg n$:

$$T(n) = S(\lg n) = \Theta(\lg n \cdot \lg\lg n).$$

3.11.5 Checking the algebra against the actual numbers

A substitution proof gives an inequality. It is reassuring — and, for a student meeting this for the first time, clarifying — to compute the recurrence exactly and see the inequality hold. **Listing 3.5** does that for both recurrences of this section.

Listing 3.5 Evaluating two recurrences exactly, and comparing against the bounds proved by substitution.

```
// verify.cpp - Solving recurrences by brute force, so the bounds proved on
// paper can be checked against the numbers they are supposed to bound.
// Compile: g++ -std=c++17 -O2 -o verify verify.cpp
#include <iostream>
#include <iomanip>
#include <map>
#include <cmath>
using namespace std;

// T(n) = 4T(n/2) + n, with T(1) = 1, evaluated only at powers of two.
double T4(long long n, map<long long,double>& memo) {
    if (n <= 1) return 1.0;
    auto it = memo.find(n);
    if (it != memo.end()) return it->second;
    double v = 4.0 * T4(n / 2, memo) + (double)n;
    memo[n] = v;
    return v;
}

// T(n) = 2T(n/2) + n, with T(1) = 1: the Merge Sort recurrence.
double T2(long long n, map<long long,double>& memo) {
    if (n <= 1) return 1.0;
    auto it = memo.find(n);
    if (it != memo.end()) return it->second;
    double v = 2.0 * T2(n / 2, memo) + (double)n;
    memo[n] = v;
    return v;
}

int main() {
    cout << "T(n) = 4T(n/2) + n,  T(1) = 1\n";
    cout << left << setw(10) << "n" << setw(16) << "T(n)"
         << setw(16) << "2n^3" << setw(20) << "2n^2 - n"
         << "T(n)/n^2\n";
    cout << string(74, '-') << "\n";
    map<long long,double> m4;
    for (long long n = 1; n <= 1024; n *= 2) {
        double t = T4(n, m4);
        cout << left << setw(10) << n << fixed << setprecision(0)
             << setw(16) << t
             << setw(16) << 2.0 * n * n * n
             << setw(20) << 2.0 * n * n - n
             << setprecision(4) << t / ((double)n * n) << "\n";
    }
```

Listing 3.5 (Continued) *Evaluating two recurrences exactly, and comparing against the bounds proved by substitution.*

```
    cout << "\nT(n) = 2T(n/2) + n,  T(1) = 1   (Merge Sort)\n";
    cout << left << setw(10) << "n" << setw(16) << "T(n)"
         << setw(20) << "n lg n + n" << "T(n)/(n lg n)\n";
    cout << string(66, '-') << "\n";
    map<long long,double> m2;
    for (long long n = 2; n <= 1024; n *= 2) {
        double t = T2(n, m2);
        double lg = log2((double)n);
        cout << left << setw(10) << n << fixed << setprecision(0)
             << setw(16) << t
             << setw(20) << n * lg + n
             << setprecision(4) << t / (n * lg) << "\n";
    }
    return 0;
}
```

The program prints:

```
T(n) = 4T(n/2) + n,  T(1) = 1
n         T(n)            2n^3            2n^2 - n            T(n)/n^2
----------------------------------------------------------------------
1         1               2               1                   1.0000
2         6               16              6                   1.5000
4         28              128             28                  1.7500
8         120             1024            120                 1.8750
16        496             8192            496                 1.9375
32        2016            65536           2016                1.9688
64        8128            524288          8128                1.9844
128       32640           4194304         32640               1.9922
256       130816          33554432        130816              1.9961
512       523776          268435456       523776              1.9980
1024      2096128         2147483648      2096128             1.9990

T(n) = 2T(n/2) + n,  T(1) = 1   (Merge Sort)
n         T(n)            n lg n + n          T(n)/(n lg n)
---------------------------------------------------------------
2         4               4                   2.0000
4         12              12                  1.5000
8         32              32                  1.3333
16        80              80                  1.2500
32        192             192                 1.2000
64        448             448                 1.1667
128       1024            1024                1.1429
256       2304            2304                1.1250
512       5120            5120                1.1111
1024      11264           11264               1.1000
```

Look at the third and fourth columns of the first table. The $O(n^3)$ bound $2n^3$ is true but wildly loose — at $n = 1024$ it is a thousand times larger than $T(n)$. The strengthened bound $2n^2 - n$ is not merely true, it is **exact**: $T(n) = 2n^2 - n$ at every single row. Our choice $c_1 = 2$, $c_2 = 1$ turned out to be the best possible. The last column, $T(n)/n^2$, climbs steadily toward 2 and never reaches it, which is $\Theta(n^2)$ made visible.

The second table is just as clean: $T(n) = n\lg n + n$ exactly, for the Merge Sort recurrence with $T(1) = 1$. The ratio $T(n)/(n\lg n)$ falls slowly toward 1, confirming $\Theta(n\lg n)$ and showing why the $+\,n$ term matters for small inputs and vanishes for large ones.

3.12 Method 2: Iteration and the Recursion Tree

The substitution method needs a guess before it can start. The recursion tree is where the guess comes from.

The idea is simply to **expand the recurrence repeatedly** — substitute the definition of T into itself, again and again — and then add up everything that falls out. Drawn as a picture, each expansion becomes a new level of a tree: the root is the original problem, its children are the subproblems, their children are the sub-subproblems, and the leaves are the base cases. The **cost written inside a node** is the non-recursive work f done at that node, not including its children.

Two numbers decide everything:

1. **How many levels are there** — that is, how many times must the recurrence be iterated before the argument reaches the boundary condition?
2. **What does each level cost in total** — the sum of the costs of all nodes at that depth?

Add the level costs, and you have $T(n)$.

3.12.1 Pure iteration: $T(n) = n + 3T(\lfloor n/4 \rfloor)$

Let us do one entirely by algebra first, with no picture, so the mechanics are clear. **Figure 3.14** shows the same unrolling written out.

$$\begin{aligned}
T(n) &= n + 3T(\lfloor n/4 \rfloor) \\
&= n + 3\big(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)\big) \\
&= n + 3\big(\lfloor n/4 \rfloor + 3\big(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor)\big)\big) \\
&= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor) \\
&\le n + \tfrac{3}{4}n + \tfrac{9}{16}n + \tfrac{27}{64}n + \cdots + 3^{\log_4 n}\, T(1) \\
&\le n \sum_{i=0}^{\infty} \left(\tfrac{3}{4}\right)^i + \Theta\big(n^{\log_4 3}\big) \\
&= 4n + o(n) \\
&= O(n).
\end{aligned}$$

Figure 3.14 Iterating $T(n) = n + 3T(\lfloor n/4 \rfloor)$ three times, collecting the geometric series $n\sum(3/4)^i$, and concluding $T(n) = O(n)$.

Expansion, one substitution per line.

$$T(n) = n + 3\,T(n/4)$$

$$= n + 3\left(\frac{n}{4} + 3\,T(n/16)\right) = n + \frac{3n}{4} + 9\,T(n/16)$$

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

$$= \cdots = n\left(1 + \frac{3}{4} + \frac{9}{16} + \frac{27}{64} + \cdots\right) + 3^{\log_4 n}\, T(1).$$

Recognizing the series. The bracket is geometric with ratio $3/4$:

$$T(n) \le n \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}).$$

Bounding it. Since $3/4 < 1$, extend the sum to infinity — it only gets bigger — and use $\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}$:

$$\sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i = \frac{1}{1 - 3/4} = 4.$$

Handling the leaf term. There are $3^{\log_4 n} = n^{\log_4 3} \approx n^{0.793}$ leaves, each costing $\Theta(1)$, so the leaf term is $\Theta(n^{0.793}) = o(n)$ — negligible beside $4n$.

Conclusion. $T(n) \le 4n + o(n) = O(n)$. ■

> **KEY INSIGHT.** Ratio below 1 ⇒ the geometric series converges ⇒ the **root** dominates ⇒ $T(n) = \Theta(f(n))$. Here $f(n) = n$, and indeed $T(n) = \Theta(n)$.

3.12.2 Drawing the tree: $T(n) = T(n/4) + T(n/2) + n^2$

This recurrence has **unequal** subproblem sizes, so the Master Theorem cannot touch it. The tree can. **Figure 3.15** shows the first two expansions: the root, labeled n^2, sprouts children of sizes $n/4$ and $n/2$; each of those sprouts its own two children, of sizes $n/16, n/8$ and $n/8, n/4$.

$$T(n) = T(n/4) + T(n/2) + n^2$$

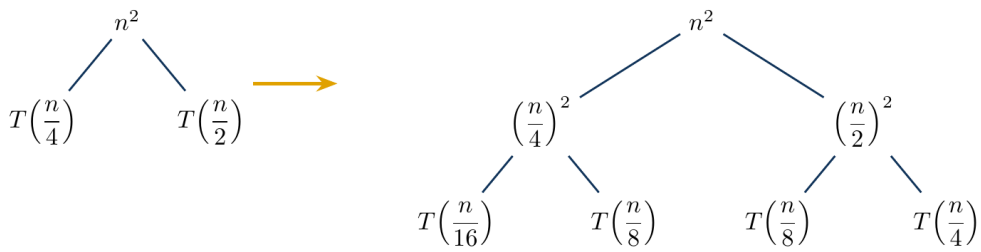


Figure 3.15 Building the recursion tree for $T(n) = T(n/4) + T(n/2) + n^2$. Each expansion replaces a $T(\cdot)$ leaf by a node carrying its own non-recursive cost and two new children.

Level costs, computed one level at a time.

Level 0. One node of size n. Cost $= n^2$.

Level 1. Two nodes, of sizes $n/4$ and $n/2$. Cost

$$\left(\frac{n}{4}\right)^2 + \left(\frac{n}{2}\right)^2 = \frac{n^2}{16} + \frac{n^2}{4} = \frac{n^2}{16} + \frac{4n^2}{16} = \frac{5n^2}{16}.$$

Level 2. Four nodes, of sizes $n/16, n/8, n/8, n/4$. Cost

$$\frac{n^2}{256}+\frac{n^2}{64}+\frac{n^2}{64}+\frac{n^2}{16}=\frac{n^2+4n^2+4n^2+16n^2}{256}=\frac{25n^2}{256}=\left(\frac{5}{16}\right)^2 n^2.$$

The pattern is unmistakable: **each level costs exactly $\frac{5}{16}$ of the level above it.** The reason is that squaring is multiplicative, so the level ratio is $(1/4)^2+(1/2)^2=\frac{5}{16}$, level after level. **Figure 3.16** shows the completed tree with the running level totals written beside it.

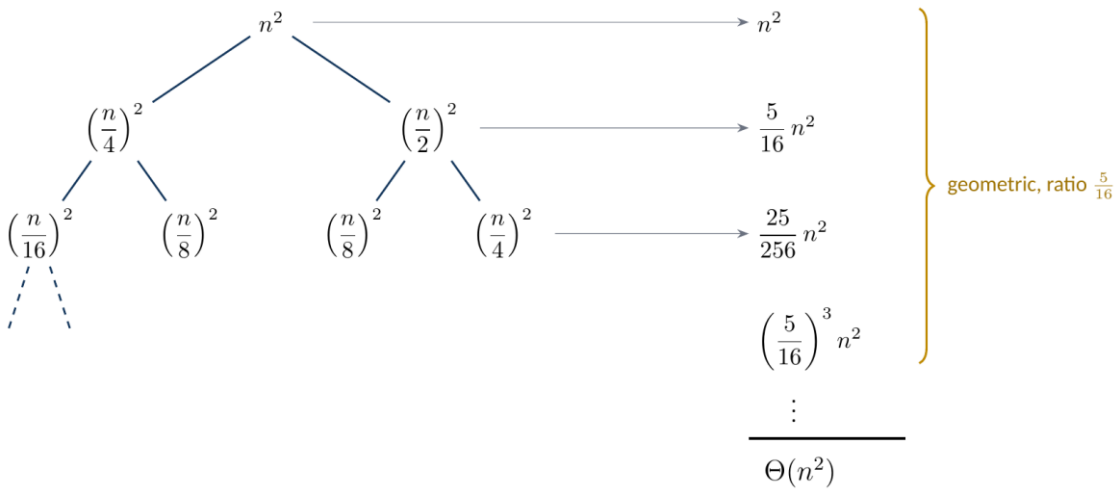


Figure 3.16 The completed recursion tree for $T(n)=T(n/4)+T(n/2)+n^2$. The level totals $n^2, \frac{5}{16}n^2, \frac{25}{256}n^2, \ldots$ form a geometric series of ratio $5/16<1$, summing to $\Theta(n^2)$.

Summing.

$$T(n)\le n^2\sum_{i=0}^{\infty}\left(\frac{5}{16}\right)^i=n^2\cdot\frac{1}{1-\frac{5}{16}}=n^2\cdot\frac{1}{\frac{11}{16}}=\frac{16}{11}n^2\approx 1.4545\,n^2.$$

And since level 0 alone already costs n^2, we have $n^2\le T(n)\le\frac{16}{11}n^2$, giving

$$T(n)=\Theta(n^2). \qquad \blacksquare$$

Note how little the recursion cost: the root pays n^2 and everything below it, all the way down, adds only another 45%.

3.12.3 When every level costs the same: $T(n)=T(n/3)+T(2n/3)+n$

Here the subproblems are lopsided — one third and two thirds — but they add back up to the whole. **Figure 3.17** is the tree.

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

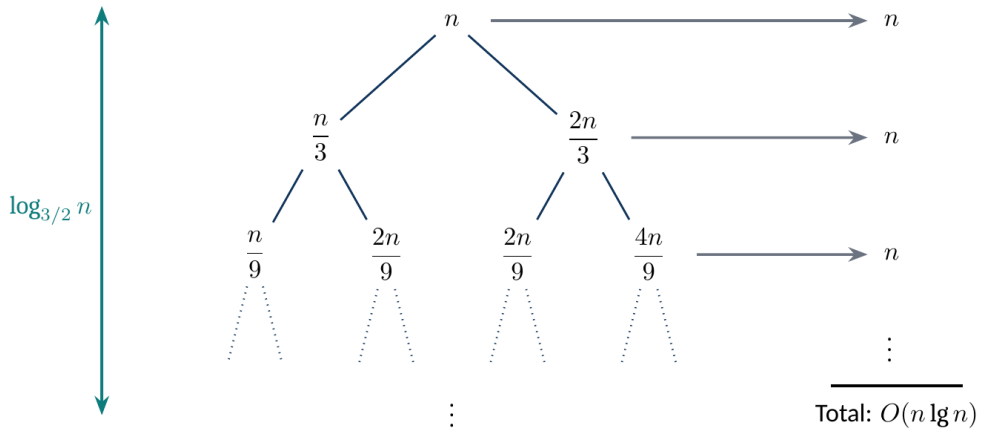


Figure 3.17 The recursion tree for $T(n) = T(n/3) + T(2n/3) + n$. Every level sums to exactly n, and the longest root-to-leaf path has $\log_{3/2} n$ levels, so the total is $O(n\lg n)$.

Level costs.

Level 0. n.

Level 1. $\frac{n}{3} + \frac{2n}{3} = n$.

Level 2. $\frac{n}{9} + \frac{2n}{9} + \frac{2n}{9} + \frac{4n}{9} = \frac{9n}{9} = n$.

Every level sums to n. That is not a coincidence: the children's sizes add up to the parent's size at every node, and the cost function $f(n) = n$ is linear, so nothing is lost in the split.

How many levels? The tree is not balanced. The shortest path down divides by 3 each time and reaches the base after $\log_3 n$ steps; the longest divides by 3/2 each time and reaches the base after $\log_{3/2} n$ steps. Using the deeper one as an upper bound:

$$T(n) \le n \cdot \log_{3/2} n = n \cdot \frac{\lg n}{\lg(3/2)} = \frac{1}{\lg 1.5}\, n\lg n \approx 1.71\, n\lg n = O(n\lg n).$$

Using the shallower one as a lower bound gives $T(n) \ge n\log_3 n = \Omega(n\lg n)$, so in fact $T(n) = \Theta(n\lg n)$.

> **KEY INSIGHT.** Level ratio exactly 1 ⇒ every level costs the same ⇒ multiply the cost of one level by the number of levels ⇒ $T(n) = \Theta(f(n)\log n)$.

3.12.4 Counting the leaves: where $n^{\log_b a}$ comes from

For the balanced form $T(n) = a\,T(n/b) + f(n)$, the tree has a rigid shape, and it is worth deriving that shape carefully because the Master Theorem's central quantity falls straight out of it.

Sizes. The root has size n. Level 1 has size n/b. Level 2 has size n/b^2. Level i has size n/b^i.

Depth. The recursion stops when the size reaches 1:

$$\frac{n}{b^i} = 1 \Leftrightarrow b^i = n \Leftrightarrow i = \log_b n.$$

So the tree has $\log_b n + 1$ levels, numbered 0 through $\log_b n$.

Node counts. The root is one node. Each node has a children, so level i holds a^i nodes.

Leaves. The bottom level is level $\log_b n$, so the number of leaves is

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

by the identity proved in §3.8.6. **Equation** Equation (3.12) **is the single most important formula in the chapter.**

Level costs. Level i holds a^i nodes, each of size n/b^i, each costing $f(n/b^i)$. So

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

The first term of Equation (3.13) is the total division-and-recombination cost across the whole tree. The second is the cost of solving all $n^{\log_b a}$ subproblems of size 1 — all the work pushed down to the leaves.

> **EXAMPLE.** Merge Sort has $a = 2$, $b = 2$, so the leaf count is $n^{\log_2 2} = n^1 = n$. Correct: sorting n items bottoms out at n single-element arrays. Binary search has $a = 1$, $b = 2$, so the leaf count is $n^{\log_2 1} = n^0 = 1$. Also correct: a single chain of calls ends at a single leaf.

3.12.5 Three shapes of tree

Everything above collapses into three cases, shown in **Table 3.6**.

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)	$\Theta(n^{\log_b a})$

Listing 3.6 builds all three shapes numerically and prints the level costs, so the ratios can be read off directly.

Listing 3.6 Summing recursion trees level by level for the three shapes of **Table 3.6**, and checking the leaf-count identity.

```
// tree.cpp - Adding up a recursion tree level by level, three different shapes.
// Compile: g++ -std=c++17 -O2 -o tree tree.cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
using namespace std;

// Expand a tree whose node of size s costs cost(s) and spawns children of
// sizes given by 'shrink'. Print the total cost of each level.
void levels(const char* title, double n, const vector<double>& shrink,
            double (*cost)(double), int maxLevel) {
    cout << title << "\n";
    vector<double> sizes = {n};
    double running = 0.0;
    for (int L = 0; L <= maxLevel && !sizes.empty(); ++L) {
        double levelCost = 0.0;
        for (double s : sizes) levelCost += cost(s);
        running += levelCost;
        cout << "  level " << setw(2) << L
             << "  nodes = " << setw(6) << sizes.size()
             << "  level cost = " << setw(14) << fixed << setprecision(2) << levelCost
             << "  running total = " << setw(14) << running << "\n";
        vector<double> next;
        for (double s : sizes)
            for (double r : shrink)
                if (s * r >= 1.0) next.push_back(s * r);
        sizes.swap(next);
    }
    cout << "\n";
}

static double square(double s) { return s * s; }
static double linear(double s) { return s; }

int main() {
    // Shape A: ratio < 1, the root dominates.
    levels("T(n) = T(n/4) + T(n/2) + n^2,  n = 1024  (level ratio 5/16 < 1)",
           1024, {0.25, 0.5}, square, 6);

    // Shape B: ratio = 1, every level costs the same.
    levels("T(n) = T(n/3) + T(2n/3) + n,   n = 1024  (level ratio 1)",
           1024, {1.0/3.0, 2.0/3.0}, linear, 6);
```

Listing 3.6 (Continued) *Summing recursion trees level by level for the three shapes of Table 3.6, and checking the leaf-count identity.*

```
    // Shape C: ratio > 1, the leaves dominate.
    levels("T(n) = 4T(n/2) + n,            n = 1024  (level ratio 2 > 1)",
           1024, {0.5, 0.5, 0.5, 0.5}, linear, 6);

    // The leaf count identity a^(log_b n) = n^(log_b a), checked numerically.
    cout << "Leaf-count identity  a^(log_b n) = n^(log_b a):\n";
    for (auto ab : vector<pair<double,double>>{{2,2},{4,2},{3,4},{9,3},{7,2}}) {
        double a = ab.first, b = ab.second, n = 4096;
        double lhs = pow(a, log(n)/log(b));
        double rhs = pow(n, log(a)/log(b));
        cout << "  a = " << a << ", b = " << b << ", n = 4096 : "
             << scientific << setprecision(6) << lhs << "  vs  " << rhs << "\n";
    }
    return 0;
}
```

The output — the program expands seven levels of each tree, which is enough to show the ratio:

```
T(n) = T(n/4) + T(n/2) + n^2,  n = 1024  (level ratio 5/16 < 1)
  level  0  nodes =      1  level cost =     1048576.00  running total =     1048576.00
  level  1  nodes =      2  level cost =      327680.00  running total =     1376256.00
  level  2  nodes =      4  level cost =      102400.00  running total =     1478656.00
  level  3  nodes =      8  level cost =       32000.00  running total =     1510656.00
  level  4  nodes =     16  level cost =       10000.00  running total =     1520656.00
  level  5  nodes =     32  level cost =        3125.00  running total =     1523781.00
  level  6  nodes =     57  level cost =         975.00  running total =     1524756.00

T(n) = T(n/3) + T(2n/3) + n,   n = 1024  (level ratio 1)
  level  0  nodes =      1  level cost =        1024.00  running total =        1024.00
  level  1  nodes =      2  level cost =        1024.00  running total =        2048.00
  level  2  nodes =      4  level cost =        1024.00  running total =        3072.00
  level  3  nodes =      8  level cost =        1024.00  running total =        4096.00
  level  4  nodes =     16  level cost =        1024.00  running total =        5120.00
  level  5  nodes =     32  level cost =        1024.00  running total =        6144.00
  level  6  nodes =     64  level cost =        1024.00  running total =        7168.00

T(n) = 4T(n/2) + n,            n = 1024  (level ratio 2 > 1)
  level  0  nodes =      1  level cost =        1024.00  running total =        1024.00
  level  1  nodes =      4  level cost =        2048.00  running total =        3072.00
  level  2  nodes =     16  level cost =        4096.00  running total =        7168.00
  level  3  nodes =     64  level cost =        8192.00  running total =       15360.00
  level  4  nodes =    256  level cost =       16384.00  running total =       31744.00
  level  5  nodes =   1024  level cost =       32768.00  running total =       64512.00
  level  6  nodes =   4096  level cost =       65536.00  running total =      130048.00

Leaf-count identity  a^(log_b n) = n^(log_b a):
  a = 2.00, b = 2.00, n = 4096 : 4.096000e+03  vs  4.096000e+03
  a = 4.000000e+00, b = 2.000000e+00, n = 4096 : 1.677722e+07  vs  1.677722e+07
  a = 3.000000e+00, b = 4.000000e+00, n = 4096 : 7.290000e+02  vs  7.290000e+02
  a = 9.000000e+00, b = 3.000000e+00, n = 4096 : 1.677722e+07  vs  1.677722e+07
  a = 7.000000e+00, b = 2.000000e+00, n = 4096 : 1.384129e+10  vs  1.384129e+10
```

Every claim of this section is visible in those numbers. In the first tree, $327680/1048576 = 0.3125 = \frac{5}{16}$ exactly, and the running total creeps toward $\frac{16}{11} \cdot 1048576 \approx 1525201$ — it has already reached 1524756 by level 6. In the second tree, every level costs exactly 1024. In the third, each level costs twice the one above, so the running total is dominated by the *last* level, not the first. And the leaf-count identity holds to six significant figures in all five test cases.

3.13 Method 3: The Master Theorem

The recursion tree of §3.12.4 always has the same shape when the recurrence has the balanced form $T(n) = a\,T(n/b) + f(n)$. Since the shape is always the same, the answer can be worked out **once**, in general, and then looked up. That lookup table is the Master Theorem.

3.13.1 The statement

> **THE MASTER THEOREM.** Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be an asymptotically positive function, and let $T(n)$ be defined on the non-negative integers by

$$T(n) = a\,T(n/b) + f(n),$$

where n/b is interpreted as either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds.

Case 1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, 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 constant $\varepsilon > 0$, **and** if $a\,f(n/b) \le c\,f(n)$ for some constant $c < 1$ and all sufficiently large n (the *regularity condition*), then $T(n) = \Theta(f(n))$.

> That is a dense paragraph. The rest of this section unpacks every piece of it.

3.13.2 What the pieces mean, in words

- a — **the branching factor.** How many recursive calls each invocation makes. Merge Sort makes two, so $a = 2$.
- b — **the shrink factor.** Each call gets an input b times smaller. Merge Sort halves, so $b = 2$.
- $f(n)$ — **the local work.** Everything the invocation does apart from the recursive calls: splitting, merging, combining. Merge Sort's merge step is $\Theta(n)$, so $f(n) = \Theta(n)$.
- $n^{\log_b a}$ — **the critical quantity, also the leaf count.** §3.12.4 proved in Equation (3.12) that the tree has $a^{\log_b n} = n^{\log_b a}$ leaves; each leaf is a base case costing $\Theta(1)$; so $n^{\log_b a}$ is *the total cost of all the leaves*.

> **THE ONE IDEA.** The Master Theorem compares two quantities: the cost at the **root**, which is $f(n)$, against the cost at the **leaves**, which is $n^{\log_b a}$. Whichever is polynomially bigger wins and becomes the answer. If they tie, neither wins and you pay for all $\lg n$ levels.

> That is the whole theorem. The three cases are: leaves win, tie, root wins.

3.13.3 The role of ε: why "polynomially"

> Cases 1 and 3 do not say $f(n)$ is *smaller* or *larger* than $n^{\log_b a}$. They say it is **polynomially** smaller or larger, and the symbol ε — a small positive constant, read "epsilon" — is how that is written down.
>
> $f(n) = O(n^{\log_b a-\varepsilon})$ means: there is some fixed positive amount ε such that f stays below n raised to a power that is a full ε *less* than the critical exponent. Being smaller by a factor of $n^{0.0001}$ counts. Being smaller merely by a factor of $\lg n$ does **not** count, because §3.8.6 showed $\lg n = o(n^{\varepsilon})$ for every $\varepsilon > 0$ — a logarithm is smaller than every positive power of n, so it can never supply the gap that ε demands.
>
> To find ε in practice, write both $f(n)$ and $n^{\log_b a}$ as powers of n and subtract the exponents.

- Case 1 needs: (exponent of f) $< \log_b a$. Then $\varepsilon = \log_b a -$ (exponent of f).
- Case 3 needs: (exponent of f) $> \log_b a$. Then $\varepsilon =$ (exponent of f) $- \log_b a$.

3.13.4 The regularity condition, in plain language

> Case 3 carries an extra hypothesis:

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

> (3.14)

What Equation (3.14) **says.** The left-hand side is the total local work done by the a children of a node. The right-hand side is a fraction of the local work done by the node itself. So the condition says: **the children's work, all of it put together, is at most a fixed fraction of the parent's work.**

Why it is needed. If that holds, then level 1 costs at most c times level 0, level 2 at most c^2 times level 0, and so on. The level costs form a geometric series with ratio at most $c < 1$, which sums to

at most $\frac{1}{1-c} f(n) = \Theta(f(n))$. Without regularity, the levels might not shrink geometrically, and the root might not dominate after all.

When it holds. Whenever $f(n) = n^k$ with $k > \log_b a$ — that is, for every polynomial f in Case 3 — because then

$$af(n/b) = a\left(\frac{n}{b}\right)^k = \frac{a}{b^k} n^k = \frac{a}{b^k} f(n),$$

and $k > \log_b a$ means $b^k > b^{\log_b a} = a$, so $c = a/b^k < 1$. **For polynomial f the condition is automatic**, which is why textbook exercises almost never see it fail.

When it fails. It fails for wildly oscillating f. The standard counterexample is

$$T(n) = T(n/2) + n\,(2 - \cos n).$$

Here $a = 1$, $b = 2$, $n^{\log_2 1} = n^0 = 1$, and $f(n) = n(2 - \cos n)$ is certainly $\Omega(n^{0+1})$, since $2 - \cos n \geq 1$ always. So the Case 3 *polynomial* condition holds. But regularity requires $f(n/2) \leq c\,f(n)$, i.e.

$$\frac{n}{2}\left(2 - \cos\frac{n}{2}\right) \leq c\,n\,(2 - \cos n).$$

Choose n where $\cos n \approx 1$ and $\cos(n/2) \approx -1$ — such n exist, arbitrarily large. Then the left side is about $\frac{n}{2} \cdot 3 = 1.5n$ and the right about $c \cdot n \cdot 1 = cn$, and $1.5n \leq cn$ forces $c \geq 1.5$. No constant $c < 1$ works. The Master Theorem does not apply.

3.13.5 The summary card

Table 3.7 is the card to memorize.

Table 3.7 The three cases of the Master Theorem for $T(n) = a\,T(n/b) + f(n)$, 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** $af(n/b) \leq cf(n)$ with $c < 1$	$T(n) = \Theta(f(n))$	the root dominates

3.13.6 Where the theorem refuses to answer: the gaps

The Master Theorem is a partial function. It does not cover every recurrence of the form $T(n) = aT(n/b) + f(n)$, and it is important to know exactly where it goes silent.

Gap between Cases 1 and 2. $f(n)$ is smaller than $n^{\log_b a}$, but not *polynomially* smaller. Example: $f(n) = n^{\log_b a}/\lg n$. Concretely,

$$T(n) = 2T(n/2) + \frac{n}{\lg n}.$$

Here $n^{\log_2 2} = n$ and $f(n) = n/\lg n$. It is not $\Theta(n)$, so Case 2 fails. Is it $O(n^{1-\varepsilon})$? That would need $\frac{n/\lg n}{n^{1-\varepsilon}} = \frac{n^\varepsilon}{\lg n}$ to stay bounded, but it tends to infinity for every $\varepsilon > 0$. So Case 1 fails too. The true answer, obtainable by a recursion tree, is $\Theta(n \lg\lg n)$.

Gap between Cases 2 and 3. $f(n)$ is larger than $n^{\log_b a}$, but not *polynomially* larger. This is the classic gap example, and the one most often answered wrongly:

$$T(n) = 2T(n/2) + n\lg n.$$

Here $a = 2$, $b = 2$, $n^{\log_2 2} = n^1 = n$, and $f(n) = n\lg n$.

Is it Case 2? We would need $n\lg n = \Theta(n)$. But $\frac{n\lg n}{n} = \lg n \to \infty$, so no.

Is it Case 3? We would need $n\lg n = \Omega(n^{1+\varepsilon})$ for some $\varepsilon > 0$, i.e. $n\lg n \geq c\, n^{1+\varepsilon}$, i.e. $\lg n \geq c\, n^\varepsilon$. But $\lg n$ grows more slowly than n^ε for every $\varepsilon > 0$. So no.

> **COMMON TRAP.** $T(n) = 2T(n/2) + n\lg n$ falls in the gap between Cases 2 and 3, and the Master Theorem **cannot** solve it. Answering "Case 3, so $\Theta(n\lg n)$" is wrong twice over: the case does not apply, and the answer is wrong. The correct answer, from a recursion tree, is $\Theta(n\lg^2 n)$ — that is, $\Theta(n(\lg n)^2)$.

Why the correct answer is $\Theta(n\lg^2 n)$. Level i has 2^i nodes of size $n/2^i$, each costing $\frac{n}{2^i}\lg\frac{n}{2^i}$, so level i costs $n\lg(n/2^i) = n(\lg n - i)$. Summing over the $\lg n$ levels:

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

(The middle step renames $j = \lg n - i$: as i runs from 0 up to $\lg n - 1$, j runs from $\lg n$ down to 1.) Equation (3.15) is the arithmetic series of §3.9.1, doing useful work again.

Other reasons the theorem does not apply.

- **$f(n)$ is not asymptotically positive.** $T(n) = 4T(n/2) - n^2$ is outside the theorem's scope.
- **a or b is not constant.** $T(n) = 2^n T(n/2) + n^n$ has a branching factor that depends on n.
- **The subproblems have different sizes.** $T(n) = T(n/3) + T(2n/3) + n$ is not of the form $aT(n/b)$; use a recursion tree (§3.12.3).
- **The recurrence subtracts instead of divides.** $T(n) = T(n-1) + n$ is not of the form $aT(n/b)$; solve it by telescoping, which gives $\Theta(n^2)$.
- **$a < 1$.** You cannot have fewer than one subproblem.

3.13.7 The five-step strategy

1. **Extract** a, b and $f(n)$ from the recurrence. Check $a \geq 1$, $b > 1$, f asymptotically positive.
2. **Compute** the critical exponent $\log_b a$, and write down $n^{\log_b a}$.
3. **Compare** $f(n)$ with $n^{\log_b a}$ by comparing exponents.
4. **Decide** the case — or recognize a gap.
5. **If Case 3, check regularity** before writing the answer.

3.14 Twenty Master Theorem Problems, Worked in Full

Everything below follows the five steps of §3.13.7. The arithmetic is shown one line at a time; nothing is left to the reader to fill in.

3.14.1 Binary search: $T(n) = T(n/2) + 1$

Binary search halves the search interval and recurses on one half, doing a constant amount of comparison work at each step. **Pseudocode 3.1** shows the algorithm in pseudocode. Its first test ends an unsuccessful search as soon as the slice $A[p..r]$ is empty, returning NIL.

Pseudocode 3.1 Recursive binary search for s in the sorted slice $A[p..r]$. One recursive call ($a = 1$) on a half-sized interval ($b = 2$), plus $\Theta(1)$ work per call, gives $T(n) = T(n/2) + \Theta(1)$.

```
BINARY-SEARCH(A, p, r, s)
    if p > r then
        return NIL
    q = floor((p + r) / 2)
    if A[q] == s then
        return q
    else if A[q] > s then
        return BINARY-SEARCH(A, p, q-1, s)
    else
        return BINARY-SEARCH(A, q+1, r, s)
```

Step 1. $a = 1, b = 2, f(n) = 1$.

Step 2. $\log_b a = \log_2 1 = 0$, because $2^0 = 1$. So $n^{\log_b a} = n^0 = 1$.

Step 3. $f(n) = 1$ and $n^{\log_b a} = 1$. They are equal, so $f(n) = \Theta(n^{\log_b a})$.

Step 4. **Case 2.**

Step 5. Not needed.

$$T(n) = \Theta(n^0 \lg n) = \Theta(\lg n). \qquad \checkmark$$

Sanity check. Halving n repeatedly until it reaches 1 takes $\lg n$ steps, and each step costs a constant. Correct.

3.14.2 $T(n) = 9T(n/3) + n$

Step 1. $a = 9, b = 3, f(n) = n$.

Step 2. $\log_3 9 = 2$, because $3^2 = 9$. So $n^{\log_b a} = n^2$.

Step 3. $f(n) = n = n^1$. The exponent 1 is less than 2. Take $\varepsilon = 2 - 1 = 1$; then $f(n) = n = O(n^{2-1})$. ✓

Step 4. **Case 1.**

$$T(n) = \Theta(n^2). \qquad \checkmark$$

Why. The leaves cost n^2 and the root only n; the bottom of the tree swamps everything.

3.14.3 Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$

Step 1. $a = 2, b = 2, f(n) = \Theta(n)$.

Step 2. $\log_2 2 = 1$. So $n^{\log_b a} = n^1 = n$.

Step 3. $f(n) = \Theta(n) = \Theta(n^{\log_b a})$. Exact tie.

Step 4. **Case 2.**

$$T(n) = \Theta(n\lg n). \qquad \checkmark$$

3.14.4 $T(n) = 3T(n/4) + n\lg n$

Step 1. $a = 3, b = 4, f(n) = n\lg n$.

Step 2. $\log_4 3 = \frac{\ln 3}{\ln 4} = \frac{1.0986}{1.3863} \approx 0.7925$. So $n^{\log_b a} \approx n^{0.7925}$.

Step 3. Is $f(n) = \Omega(n^{0.7925+\varepsilon})$? Try $\varepsilon = 0.2$, so the target is $n^{0.9925}$. Then

$$\frac{n\lg n}{n^{0.9925}} = n^{0.0075}\lg n \to \infty,$$

so $n\lg n$ is indeed $\Omega(n^{0.9925})$. ✓

Step 4. **Case 3 candidate.**

Step 5 — regularity. We need $af(n/b) \le cf(n)$ with $c < 1$:

$$3f\left(\frac{n}{4}\right) = 3 \cdot \frac{n}{4}\lg\frac{n}{4} = \frac{3n}{4}(\lg n - 2) \le \frac{3n}{4}\lg n = \frac{3}{4}f(n).$$

So $c = 3/4 < 1$. ✓

$$T(n) = \Theta(n\lg n). \qquad \checkmark$$

3.14.5 $T(n) = 4T(n/2) + n^3$

Step 1. $a = 4, b = 2, f(n) = n^3$.

Step 2. $\log_2 4 = 2$. So $n^{\log_b a} = n^2$.

Step 3. Exponent of f is $3 > 2$. Take $\varepsilon = 1$: $f(n) = n^3 = \Omega(n^{2+1})$. ✓

Step 4. **Case 3 candidate.**

Step 5 — regularity.

$$4f\left(\frac{n}{2}\right) = 4\left(\frac{n}{2}\right)^3 = 4 \cdot \frac{n^3}{8} = \frac{n^3}{2} = \frac{1}{2}f(n),$$

so $c = 1/2 < 1$. ✓

$$T(n) = \Theta(n^3). \qquad \checkmark$$

3.14.6 The gap: $T(n) = 2T(n/2) + n\lg n$

Worked in full in §3.13.6. $a = 2, b = 2, n^{\log_2 2} = n, f(n) = n\lg n$. Neither Case 2 nor Case 3 applies: $n\lg n$ is larger than n but not *polynomially* larger.

The Master Theorem does not apply. By recursion tree: $T(n) = \Theta(n\lg^2 n)$.

3.14.7 $T(n) = 4T(n/2) + n$

Step 1. $a = 4, b = 2, f(n) = n$.

Step 2. $\log_2 4 = 2$, so $n^{\log_b a} = n^2$.

Step 3. Exponent of f is $1 < 2$; take $\varepsilon = 1$, giving $f(n) = O(n^{2-1})$. ✓

Step 4. **Case 1.**

$$T(n) = \Theta(n^2). \qquad ✓$$

This is the recurrence we proved by substitution in §3.11.3, and **Listing 3.5** showed $T(n) = 2n^2 - n$ exactly. The Master Theorem delivers the same answer in four lines instead of a page.

3.14.8 $T(n) = 4T(n/2) + n^2$

Step 1. $a = 4, b = 2, f(n) = n^2$. *Step 2.* $n^{\log_2 4} = n^2$.

Step 3. $f(n) = n^2 = \Theta(n^2)$. Tie. *Step 4.* **Case 2.**

$$T(n) = \Theta(n^2 \lg n). \qquad ✓$$

Compare with §3.14.7 and §3.14.5: the same a and b, three different f, three different cases. This trio is worth memorizing as a set.

3.14.9 $T(n) = 8T(n/2) + n^2$

Step 1. $a = 8, b = 2, f(n) = n^2$.

Step 2. $\log_2 8 = 3$, so $n^{\log_b a} = n^3$.

Step 3. Exponent $2 < 3$; take $\varepsilon = 1$. **Case 1.**

$$T(n) = \Theta(n^3). \qquad ✓$$

This is naive divide-and-conquer matrix multiplication: eight recursive multiplications of half-sized matrices, plus $\Theta(n^2)$ additions. It is no better than the schoolbook triple loop.

3.14.10 Strassen: $T(n) = 7T(n/2) + n^2$

Step 1. $a = 7, b = 2, f(n) = n^2$.

Step 2. $\log_2 7 = \frac{\ln 7}{\ln 2} = \frac{1.9459}{0.6931} \approx 2.807$. So $n^{\log_b a} \approx n^{2.807}$.

Step 3. Exponent of f is $2 < 2.807$. Take $\varepsilon = 0.8$, so the target is $n^{2.007}$, and $n^2 = O(n^{2.007})$. ✓

Step 4. **Case 1.**

$$T(n) = \Theta(n^{\lg 7}) \approx \Theta(n^{2.807}). \qquad ✓$$

Strassen's insight was to replace eight multiplications with seven at the cost of more additions — and since $f(n)$ was already in Case 1, the extra additions cost nothing asymptotically while the reduced a cuts the exponent from 3 to 2.807.

3.14.11 $T(n) = 8T(n/2) + n^3$

Step 1. $a = 8, b = 2, f(n) = n^3$. *Step 2.* $n^{\log_2 8} = n^3$.

Step 3. $f(n) = n^3 = \Theta(n^3)$. Tie. *Step 4.* **Case 2.**

$$T(n) = \Theta(n^3 \lg n). \qquad \checkmark$$

3.14.12 $T(n) = 2T(n/4) + \sqrt{n}$

Step 1. $a = 2, b = 4, f(n) = \sqrt{n} = n^{1/2}$.

Step 2. $\log_4 2 = 1/2$, because $4^{1/2} = 2$. So $n^{\log_b a} = n^{1/2} = \sqrt{n}$.

Step 3. $f(n) = \sqrt{n} = \Theta(\sqrt{n})$. Tie. *Step 4.* **Case 2.**

$$T(n) = \Theta(\sqrt{n} \lg n). \qquad \checkmark$$

3.14.13 $T(n) = 3T(n/3) + n/2$

Step 1. $a = 3, b = 3, f(n) = n/2$.

Step 2. $\log_3 3 = 1$, so $n^{\log_b a} = n$.

Step 3. $f(n) = n/2 = \Theta(n)$ — the constant $1/2$ is absorbed. Tie. *Step 4.* **Case 2.**

$$T(n) = \Theta(n \lg n). \qquad \checkmark$$

The lesson. Constant factors inside $f(n)$ never change the case. Do not let the $1/2$ distract you.

3.14.14 $T(n) = 16T(n/4) + n^2$

Step 1. $a = 16, b = 4, f(n) = n^2$.

Step 2. $\log_4 16 = 2$, because $4^2 = 16$. So $n^{\log_b a} = n^2$.

Step 3. Tie. *Step 4.* **Case 2.**

$$T(n) = \Theta(n^2 \lg n). \qquad \checkmark$$

3.14.15 $T(n) = T(n/2) + n$

Step 1. $a = 1, b = 2, f(n) = n$.

Step 2. $\log_2 1 = 0$, so $n^{\log_b a} = 1$.

Step 3. Exponent of f is $1 > 0$; take $\varepsilon = 1$, giving $f(n) = \Omega(n^{0+1})$. ✓

Step 4. **Case 3 candidate.**

Step 5 — regularity. $1 \cdot f(n/2) = \frac{n}{2} = 1/2\, f(n)$, so $c = 1/2 < 1$. ✓

$$T(n) = \Theta(n). \qquad \checkmark$$

Sanity check. The work is $n + \frac{n}{2} + \frac{n}{4} + \cdots < 2n$. Correct.

3.14.16 The other gap: $T(n) = 2T(n/2) + n/\lg n$

Step 1. $a = 2, b = 2, f(n) = n/\lg n$. *Step 2.* $n^{\log_2 2} = n$.

Step 3. Is $f = \Theta(n)$? No: $f(n)/n = 1/\lg n \to 0$. Is $f = O(n^{1-\varepsilon})$? That needs $\frac{n/\lg n}{n^{1-\varepsilon}} = \frac{n^{\varepsilon}}{\lg n}$ bounded, but it diverges for every $\varepsilon > 0$.

Gap between Cases 1 and 2 — the theorem does not apply. True answe

3.14.17 $T(n) = 5T(n/2) + n^2$

Step 1. $a = 5, b = 2, f(n) = n^2$.

Step 2. $\log_2 5 = \frac{\ln 5}{\ln 2} = \frac{1.6094}{0.6931} \approx 2.3219$.

Step 3. Exponent $2 < 2.3219$; take $\varepsilon = 0.3$, target $n^{2.0219}$, and $n^2 = O(n^{2.0219})$. ✓ *Step 4.* **Case 1.**

$$T(n) = \Theta(n^{\lg 5}) \approx \Theta(n^{2.322}). \qquad \checkmark$$

3.14.18 $T(n) = T(2n/3) + 1$

Step 1. $a = 1, b = 3/2, f(n) = 1$.

Step 2. $\log_{3/2} 1 = 0$, so $n^{\log_b a} = 1$.

Step 3. $f(n) = 1 = \Theta(1)$. Tie. *Step 4.* **Case 2.**

$$T(n) = \Theta(\lg n). \qquad \checkmark$$

Note. b need not be an integer. Shrinking by any constant factor greater than 1 gives a logarithmic number of levels.

3.14.19 Regularity earning its keep: $T(n) = 2T(n/2) + n^2/\lg n$

Step 1. $a = 2, b = 2, f(n) = n^2/\lg n$. *Step 2.* $n^{\log_2 2} = n$.

Step 3. Is $f = \Omega(n^{1+\varepsilon})$? With $\varepsilon = 1/2$:

$$\frac{n^2/\lg n}{n^{1.5}} = \frac{n^{0.5}}{\lg n} \to \infty. \checkmark$$

Step 4. **Case 3 candidate.**

Step 5 — regularity, computed carefully.

$$2f\left(\frac{n}{2}\right) = 2 \cdot \frac{(n/2)^2}{\lg(n/2)} = 2 \cdot \frac{n^2/4}{\lg n - 1} = \frac{n^2}{2(\lg n - 1)}.$$

Divide by $f(n) = n^2/\lg n$:

$$\frac{2f(n/2)}{f(n)} = \frac{n^2}{2(\lg n - 1)} \cdot \frac{\lg n}{n^2} = \frac{\lg n}{2(\lg n - 1)} \to \frac{1}{2}.$$

For $n \geq 16$ we have $\lg n \geq 4$, so $\frac{\lg n}{2(\lg n - 1)} \leq \frac{4}{2 \cdot 3} = 2/3 < 1$. Take $c = 2/3$. ✓

$$T(n) = \Theta\left(\frac{n^2}{\lg n}\right). \qquad ✓$$

The lesson. When f is not a pure power of n, regularity is a genuine calculation, not a formality.

3.14.20 Regularity failing: $T(n) = T(n/2) + n(2 - \cos n)$

Worked in §3.13.4. The polynomial part of Case 3 holds, but no constant $c < 1$ satisfies $f(n/2) \leq cf(n)$, because f oscillates. **The Master Theorem does not apply**, even though the recurrence has exactly the right shape. This is the standard example showing that the regularity condition is not redundant.

3.14.21 Checking all of them by machine

Listing 3.7 implements the five-step strategy directly: it accepts a, b and an f of the form $n^p(\lg n)^q$, computes $\log_b a$, compares p with it, names the case, and checks regularity.

Listing 3.7 A Master Theorem solver in C++, applied to eighteen recurrences.

```cpp
// master.cpp - A Master Theorem solver for T(n) = a T(n/b) + f(n)
// with f(n) = n^p (lg n)^q. It computes the critical exponent log_b(a),
// compares it with p, names the case, checks regularity, and prints Theta(.).
// Compile: g++ -std=c++17 -O2 -o master master.cpp
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>
#include <vector>
#include <cmath>
using namespace std;

struct Rec { double a, b, p; int q; string name; };

// Print a real exponent compactly: "2" when it is an integer, "2.807" otherwise.
static string expo(double e) {
    ostringstream os;
    if (fabs(e - llround(e)) < 1e-9) os << llround(e);
    else os << fixed << setprecision(3) << e;
    return os.str();
}

// Render n^e, collapsing n^0 to 1 and n^1 to n.
static string npow(double e) {
    if (fabs(e) < 1e-9) return "1";
    if (fabs(e - 1.0) < 1e-9) return "n";
    return "n^" + expo(e);
}

static string lgpow(int q) {
    if (q == 0) return "";
    if (q == 1) return " lg n";
    return " lg^" + to_string(q) + " n";
}

int main() {
    vector<Rec> rs = {
        {1, 2,   0, 0, "T(n) = T(n/2) + 1"},
        {9, 3,   1, 0, "T(n) = 9T(n/3) + n"},
        {2, 2,   1, 0, "T(n) = 2T(n/2) + n"},
        {3, 4,   1, 1, "T(n) = 3T(n/4) + n lg n"},
        {4, 2,   3, 0, "T(n) = 4T(n/2) + n^3"},
        {2, 2,   1, 1, "T(n) = 2T(n/2) + n lg n"},
        {4, 2,   1, 0, "T(n) = 4T(n/2) + n"},
        {4, 2,   2, 0, "T(n) = 4T(n/2) + n^2"},
        {8, 2,   2, 0, "T(n) = 8T(n/2) + n^2"},
```

Listing 3.7 (Continued) *A Master Theorem solver in C++, applied to eighteen recurrences.*

```
        {7, 2,   2, 0, "T(n) = 7T(n/2) + n^2"},
        {8, 2,   3, 0, "T(n) = 8T(n/2) + n^3"},
        {2, 4, 0.5, 0, "T(n) = 2T(n/4) + sqrt(n)"},
        {3, 3,   1, 0, "T(n) = 3T(n/3) + n/2"},
        {16,4,   2, 0, "T(n) = 16T(n/4) + n^2"},
        {1, 2,   1, 0, "T(n) = T(n/2) + n"},
        {2, 2,   1,-1, "T(n) = 2T(n/2) + n/lg n"},
        {5, 2,   2, 0, "T(n) = 5T(n/2) + n^2"},
        {1, 1.5, 0, 0, "T(n) = T(2n/3) + 1"},
    };

    cout << left << setw(30) << "recurrence" << setw(10) << "log_b a"
         << setw(16) << "f(n)" << setw(10) << "case" << "T(n)\n";
    cout << string(96, '-') << "\n";

    const double EPS = 1e-9;
    for (const Rec& r : rs) {
        double crit = log(r.a) / log(r.b);
        string f = npow(r.p) + lgpow(r.q);
        if (r.q == -1) f = "n/lg n";
        string kase, ans;

        if (r.p < crit - EPS) {                    // polynomially smaller
            kase = "1";
            ans = "Theta(" + npow(crit) + ")";
        } else if (fabs(r.p - crit) < EPS) {       // same polynomial order
            if (r.q == 0) { kase = "2"; string base = npow(crit);
                            ans = "Theta(" + (base == "1" ? "lg n" : base + " lg n") + ")";
}
            else if (r.q > 0) { kase = "gap 2-3"; ans = "Theta(" + npow(crit) + lgpow(r.q +
1) + ") by other means"; }
            else { kase = "gap <2"; ans = "not covered (answer Theta(n lg lg n))"; }
        } else {                                   // polynomially larger
            double c = r.a / pow(r.b, r.p);        // regularity constant
            if (c < 1.0 - 1e-12) {
                kase = "3";
                ostringstream os; os << fixed << setprecision(3) << c;
                ans = "Theta(" + f + ")   [regularity c = " + os.str() + " < 1]";
            } else { kase = "3?"; ans = "regularity fails"; }
        }
        cout << left << setw(30) << r.name << setw(10) << expo(crit)
             << setw(16) << f << setw(10) << kase << ans << "\n";
    }
    return 0;
}
```

Its output is exactly the table of answers derived by hand above:

```
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       0.792     n lg n          3         Theta(n lg n)
[regularity c = 0.750 < 1]
T(n) = 4T(n/2) + n^3          2         n^3             3         Theta(n^3)   [regularity
c = 0.500 < 1]
T(n) = 2T(n/2) + n lg n       1         n lg n          gap 2-3   Theta(n lg^2 n) by other
means
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^2.807)
T(n) = 8T(n/2) + n^3          3         n^3             2         Theta(n^3 lg n)
T(n) = 2T(n/4) + sqrt(n)      0.500     n^0.500         2         Theta(n^0.500 lg n)
T(n) = 3T(n/3) + n/2          1         n               2         Theta(n lg 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)   [regularity c
= 0.500 < 1]
T(n) = 2T(n/2) + n/lg n       1         n/lg n          gap <2    not covered (answer
Theta(n lg lg n))
T(n) = 5T(n/2) + n^2          2.322     n^2             1         Theta(n^2.322)
T(n) = T(2n/3) + 1            0         1               2         Theta(lg n)
```

Every hand-computed regularity constant reappears: $3/4 = 0.750$ in §3.14.4 and $1/2 = 0.500$ in §3.14.5 and §3.14.15. The program also confirms both gaps, and — importantly — it *knows* they are gaps rather than silently returning a wrong case. **Table 3.8** collects the results.

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	$\Theta(n\lg n)$
$T(n) = 16T(n/4) + n^2$	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	$\Theta(n^2/\lg n)$
$T(n) = T(n/2) + n(2 - \cos n)$	0	oscillating	none	regularity fails

3.15 Merge Sort, Derived from Nothing

We now have every tool the chapter promised. Let us use them on a real algorithm, from the first idea to a running C++ program and a proof that it is correct and fast.

3.15.1 The divide-and-conquer blueprint

> **DEFINITION.** A **divide-and-conquer** algorithm solves a problem in three steps. **DIVIDE** the problem into smaller subproblems *of the same kind*. **CONQUER** the subproblems by solving them recursively; if a subproblem is small enough, solve it directly. **COMBINE** the subproblem solutions into a solution for the original problem.

Sun Tzu's remark in *The Art of War* — *"the control of a large force is the same principle as the control of a few men: it is merely a question of dividing up their numbers"* — is the entire strategy in one sentence.

Apply it to sorting.

- **Divide.** Split the array in half. Nothing clever: just cut it at the midpoint. This costs $\Theta(1)$ if we pass indices rather than copying.
- **Conquer.** Sort each half by calling ourselves. An array of one element is already sorted, so that is the base case.

- **Combine.** Take two sorted halves and interleave them into one sorted whole. This is the **merge**, and it is where all the real work lives.

3.15.2 The algorithm

Pseudocode 3.2 gives the algorithm in pseudocode.

Pseudocode 3.2 Merge Sort on the slice $A[p..r]$, with MERGE described in words; **Listing 3.8** 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].
```

Here p is the index of the first element of the slice, r the index of the last, and q the midpoint. The test `if p < r` is the base case in disguise: when $p = r$ the slice holds one element and there is nothing to do; when $p > r$ it is empty.

3.15.3 A complete trace on $A = [1,5,2,4,6,3,2,6]$

Divide phase (top-down splitting).

```
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
```

Combine phase (bottom-up merging).

```
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]
```

Follow one merge slowly, the last one. The left list is [1,2,4,5] and the right is [2,3,6,6]. Compare the fronts: $1 < 2$, take 1. Now [2,4,5] against [2,3,6,6]: the fronts tie at 2, and we take the **left** one (this choice is what makes the sort *stable*). Now [4,5] against [2,3,6,6]: $2 < 4$, take 2. Then $3 < 4$, take 3. Then $4 < 6$, take 4; then $5 < 6$, take 5. The left list is exhausted, so copy the rest of the right: 6,6. Result: [1,2,2,3,4,5,6,6]. ✓

3.15.4 Proving MERGE correct with a loop invariant

The merge loop repeatedly copies the smaller of two front elements. Why does that produce a sorted array? Here is the proof.

Let L be a copy of the sorted left half $A[p..q]$ and R a copy of the sorted right half $A[q+1..r]$. Let i index the next unused element of L, j the next unused element of R, and k the next position to fill in A. Initially $i = j = 0$ and $k = p$.

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

Initialization. Before the first iteration, $k = p$, so $A[p..k-1]$ is empty — it contains the 0 smallest elements, vacuously sorted, and (a) holds. Since $i = j = 0$ and both L and R are sorted, $L[0]$ and $R[0]$ are their smallest elements, so (b) holds. ✓

Maintenance. Suppose the invariant holds at the top of an iteration. Two cases.

If $L[i] \leq R[j]$, then $L[i]$ is the smallest element not yet copied — it beats everything remaining in L because L is sorted, and it beats everything remaining in R because it is $\leq R[j]$, which is the smallest of those. The line `A[k++] = L[i++]` puts it at position k. Now $A[p..k]$ holds the $k - p + 1$ smallest elements in sorted order, because we appended an element that is at least as large as every element already there. Incrementing i makes $L[i]$ the new smallest unused element of L. So (a) and (b) hold with k and i advanced by one. ✓

If $L[i] > R[j]$, the same argument applies with the roles of L and R exchanged. ✓

Termination. The loop ends when i reaches $|L|$ or j reaches $|R|$ — say L is exhausted. By (a), $A[p..k-1]$ holds the $k - p$ smallest elements in sorted order; the elements still missing are exactly $R[j..]$, which are sorted and all at least as large as everything already placed (because everything already placed was, at the time it was placed, no larger than the then-front of R). The two clean-up loops append them in order. The array $A[p..r]$ is therefore sorted and is a permutation of the original. That is the postcondition. ■

Termination of the loop itself. Each iteration increments exactly one of i or j, both bounded above, so the loop runs at most $|L| + |R| = r - p + 1$ times and must stop. Combined with partial correctness, MERGE is **totally correct**, and it runs in $\Theta(n)$ time on a slice of length n, since it does a constant amount of work per element copied.

Correctness of MERGE-SORT itself then follows by strong induction on the slice length: slices of length 1 are sorted (base case); a slice of length $m > 1$ is split into two strictly shorter slices, each sorted by the inductive hypothesis, and MERGE combines two sorted slices into a sorted whole. ■

3.15.5 The recurrence, and three ways to solve it

Divide costs $\Theta(1)$; conquer costs $2T(n/2)$; combine costs $\Theta(n)$. Total:

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

Recurrence Equation (3.16) is exactly **Figure 3.10**.

By recursion tree. Write the combine cost as cn for a constant c. The root costs cn. Level 1 has two nodes of size $n/2$, costing $2 \cdot c\frac{n}{2} = cn$. Level 2 has four nodes of size $n/4$, costing $4 \cdot c\frac{n}{4} = cn$. **Every level costs** cn**.** The tree has $\lg n + 1$ levels, because halving n reaches 1 after $\lg n$ halvings. Therefore

$$T(n) = cn(\lg n + 1) = cn\lg n + cn = \Theta(n\lg n).$$

By the Master Theorem. $a = 2$, $b = 2$, $n^{\log_2 2} = n$, $f(n) = \Theta(n)$: Case 2, so $T(n) = \Theta(n\lg n)$ (§3.14.3).

By substitution. Guess $T(n) \leq cn\lg n$ and verify:

$$T(n) = 2T(n/2) + n \le 2 \cdot c\frac{n}{2}\lg\frac{n}{2} + n = cn(\lg n - 1) + n$$
$$= cn\lg n - cn + n \le cn\lg n \text{ when } c \ge 1.$$

Three methods, one answer. And **Listing 3.5** already showed the exact solution is $T(n) = n\lg n + n$ when $T(1) = 1$ — matching $cn\lg n + cn$ with $c = 1$ precisely.

3.15.6 The program

Listing 3.8 is the complete, compilable implementation, with the trace of §3.15.3 printed by the program itself and a comparison count measured against the $n\lg n$ prediction.

Listing 3.8 Merge Sort in C++, with a printed trace and a measured comparison count.

```
// 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 3.8 (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;
}
```

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 trace reproduces §3.15.3 line for line, ending at [1,2,2,3,4,5,6,6]. The table underneath is the asymptotic claim confirmed by measurement: as n grows by a factor of four thousand, the ratio of actual comparisons to $n\lg n$ stays pinned between 0.73 and 0.93 — never wandering off, which is exactly what $\Theta(n\lg n)$ means. (The ratio sits below 1 because a merge of two lists of length m each finishes after at most $2m - 1$ comparisons, not $2m$: the last block is copied without comparing.)

3.15.7 What Merge Sort costs, and what it buys

Merge Sort is $\Theta(n\lg n)$ in the **best, average and worst** cases alike — it always splits down the middle and always merges the same way, so there is no input that makes it slow. That is a real guarantee, and Insertion Sort cannot offer it.

The price is space. The merge step needs somewhere to hold the two halves, so Merge Sort uses $\Theta(n)$ extra memory and is **not in place**. Chapter 5 introduces Heapsort, which is $\Theta(n\lg n)$ *and* in place, and Quicksort, which is $\Theta(n^2)$ in the worst case but usually faster than both in practice. Merge Sort keeps one advantage over both: it is **stable**, meaning equal elements keep their original relative order — which is exactly why the `<=` in the merge comparison matters, and why Merge Sort is the algorithm of choice when records carry more than the key you are sorting on.

3.16 Profitina in Practice

Profitina is an **AI Visibility (GEO)** platform: it scans a brand's website, scores how ready that site is to be *cited by AI answer engines*, and explains each signal it measured. Generative Engine Optimization is to answer engines what search engine optimization was to lists of links. Everything in this chapter is load-bearing in that product, and the shape of the problem — never the proprietary scoring internals — is worth describing.

The scan is a recurrence, not a loop. A site is a tree: a home page links to section pages, which link to article pages. A scan of a site with n pages visits each page, extracts its text and structured data, and recurses into the pages it links to. If the crawl frontier is split into a shards handled independently and each shard is b times smaller, the cost obeys $T(n) = a\,T(n/b) + f(n)$, where $f(n)$ is the per-batch work of fetching and parsing. The engineering question — *does adding more shards actually help?* — is exactly the question of which Master Theorem case you are in. If $f(n)$ dominates (Case 3), sharding buys nothing, because the bottleneck is the parsing, not the recursion. If the leaves dominate (Case 1), sharding is the whole game.

Sorting appears everywhere, and it must be $n\lg n$. The historical analytics dashboard shows a weekly score trend per brand over a growing table of measurements. Rendering a trend means ordering measurements by timestamp; deduplicating near-identical pages means sorting by content hash and scanning for runs of equal keys; ranking a brand's worst-scoring pages means sorting by score. Each of those is an $O(n\lg n)$ sort followed by an $O(n)$ pass. Doing any of them with an $O(n^2)$ method would be invisible during development, on a test brand with forty pages, and catastrophic in production on a brand with forty thousand.

Stability is a product requirement, not a nicety. When two pages score identically, the dashboard must list them in a stable, reproducible order, or a customer refreshing the page sees the rows jump around and loses trust in the number. Merge Sort's stability — the `<=` of §3.15.6 — is what delivers that.

The re-scan scheduler is a priority-queue problem. Every brand is re-scanned automatically at a cadence that depends on its subscription tier. Choosing which brand to scan next is "extract

the minimum next-due time", which Chapter 5's binary heap does in $O(\lg n)$. Scanning a list of due brands by repeatedly searching the whole list would be $O(n)$ per pick and $O(n^2)$ per cycle.

And the discipline of the gap case. Real measured cost curves rarely land exactly on a textbook case. A step whose cost is $n\lg n$ inside a recurrence that halves — Case-2-versus-Case-3 gap territory — is genuinely $\Theta(n\lg^2 n)$, not $\Theta(n\lg n)$, and the difference shows up as a capacity-planning error if you round it to the nearer case. Knowing where the theorem stops answering is as useful as knowing what it answers.

> **Two rules when this material touches a real product.** First, Profitina advises; the brand owner decides. A score is a measurement with an explanation attached, never a guarantee of being cited. Second, anything that touches personal data is governed by **UU PDP 27/2022** — explicit consent, data export, full account purge.

3.16.1 The worked case study: Profitina Laundry

The runnable case study in this book is always **Profitina Laundry**, a teaching business, not the real product. The shop takes in baskets of washing across the morning, each with a ticket number, a promised pickup time, and a weight. The counter clerk enters them in the order customers walk in, which is not the order in which they must be finished.

Two questions the owner actually asks:

1. *In what order should we run the machines?* — Sort the baskets by promised pickup time. That is Merge Sort, $\Theta(n\lg n)$.
2. *How badly out of order is the queue as entered?* — Count the **inversions**: pairs of baskets where an earlier-entered basket is due *later* than a later-entered one. A queue with zero inversions is already in the right order; a queue with the maximum $\binom{n}{2}$ inversions is exactly backward. This number tells the owner how much the counter's intake order is costing them.

The naive way to count inversions is to compare every pair: $\binom{n}{2} = \Theta(n^2)$. But the merge step already discovers inversions for free. When the merge takes an element from the right half, every element still remaining in the left half is larger than it — and each of those is an inversion. So we add $|L| - i$ to a counter, in $O(1)$, and get all of them. The recurrence is unchanged:

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n\lg n),$$

so we count inversions as fast as we sort. **Listing 3.9** does both in one pass.

Listing 3.9 Profitina Laundry: Merge Sort by promised pickup time, counting inversions in the same $\Theta(n\lg n)$ pass.

```cpp
// laundry.cpp - Profitina Laundry: order the day's baskets by promised
// pickup time with Merge Sort, and, in the same divide-and-conquer pass,
// count how many pairs are currently out of order (inversions).
// Both run in T(n) = 2T(n/2) + Theta(n) = Theta(n lg n).
// Compile: g++ -std=c++17 -O2 -o laundry laundry.cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <sstream>
using namespace std;

struct Order {
    string ticket;      // e.g. "PL-014"
    int    minutes;     // promised pickup, minutes after 06:00
    double kilograms;
};

static string clockOf(int minutes) {          // 0 -> "06:00"
    int h = 6 + minutes / 60, m = minutes % 60;
    ostringstream os;
    os << setw(2) << setfill('0') << h << ':' << setw(2) << setfill('0') << m;
    return os.str();
}

// Merge A[p..q] with A[q+1..r], both already sorted by 'minutes',
// returning the number of inversions removed: pairs (i, j) with i in the
// left half, j in the right half, and A[i].minutes > A[j].minutes.
long long mergeCount(vector<Order>& A, int p, int q, int r) {
    vector<Order> L(A.begin() + p, A.begin() + q + 1);
    vector<Order> R(A.begin() + q + 1, A.begin() + r + 1);
    size_t i = 0, j = 0; int k = p; long long inv = 0;
    while (i < L.size() && j < R.size()) {
        if (L[i].minutes <= R[j].minutes) A[k++] = L[i++];
        else {
            // L[i..] all exceed R[j], so L.size()-i inversions vanish at once.
            inv += (long long)(L.size() - i);
            A[k++] = R[j++];
        }
    }
    while (i < L.size()) A[k++] = L[i++];
    while (j < R.size()) A[k++] = R[j++];
    return inv;
}
```

Listing 3.9 (Continued) *Profitina Laundry: Merge Sort by promised pickup time, counting inversions in the same \Theta(n\lg n) pass.*

```
long long sortCount(vector<Order>& A, int p, int r) {
    if (p >= r) return 0;
    int q = p + (r - p) / 2;
    long long inv = sortCount(A, p, q);
    inv += sortCount(A, q + 1, r);
    inv += mergeCount(A, p, q, r);
    return inv;
}

int main() {
    vector<Order> day = {
        {"PL-014", 300, 4.5}, {"PL-002",  90, 2.0}, {"PL-021", 420, 7.25},
        {"PL-008", 150, 3.0}, {"PL-017", 240, 5.5}, {"PL-005", 120, 1.5},
        {"PL-030", 480, 9.0}, {"PL-011", 210, 6.0}, {"PL-026", 360, 3.75},
        {"PL-019", 270, 2.25}, {"PL-003",  60, 8.0}, {"PL-023", 330, 5.0}
    };

    cout << "Queue as the counter entered it (" << day.size() << " baskets)\n";
    for (const Order& o : day)
        cout << "  " << o.ticket << "  due " << clockOf(o.minutes)
             << "  " << fixed << setprecision(2) << setw(5) << o.kilograms << " kg\n";

    long long inversions = sortCount(day, 0, (int)day.size() - 1);

    cout << "\nQueue after Merge Sort, earliest promise first\n";
    double total = 0.0;
    for (const Order& o : day) {
        total += o.kilograms;
        cout << "  " << o.ticket << "  due " << clockOf(o.minutes)
             << "  " << setw(5) << o.kilograms << " kg\n";
    }
    cout << "\ntotal load          : " << total << " kg\n";
    cout << "out-of-order pairs  : " << inversions
         << "  (maximum possible = " << day.size() * (day.size() - 1) / 2 << ")\n";
    return 0;
}
```

The output:

```
Queue as the counter entered it (12 baskets)
  PL-014  due 11:00   4.50 kg
  PL-002  due 07:30   2.00 kg
  PL-021  due 13:00   7.25 kg
  PL-008  due 08:30   3.00 kg
  PL-017  due 10:00   5.50 kg
  PL-005  due 08:00   1.50 kg
  PL-030  due 14:00   9.00 kg
  PL-011  due 09:30   6.00 kg
  PL-026  due 12:00   3.75 kg
  PL-019  due 10:30   2.25 kg
  PL-003  due 07:00   8.00 kg
  PL-023  due 11:30   5.00 kg

Queue after Merge Sort, earliest promise first
  PL-003  due 07:00   8.00 kg
  PL-002  due 07:30   2.00 kg
  PL-005  due 08:00   1.50 kg
  PL-008  due 08:30   3.00 kg
  PL-011  due 09:30   6.00 kg
  PL-017  due 10:00   5.50 kg
  PL-019  due 10:30   2.25 kg
  PL-014  due 11:00   4.50 kg
  PL-023  due 11:30   5.00 kg
  PL-026  due 12:00   3.75 kg
  PL-021  due 13:00   7.25 kg
  PL-030  due 14:00   9.00 kg
```

and then the summary:

```
total load          : 57.75 kg
out-of-order pairs  : 32  (maximum possible = 66)
```

Thirty-two inversions out of a possible sixty-six: the intake order is about as disordered as a random shuffle, which would produce 33 inversions on average (each of the 66 pairs is out of order with probability one half). The owner now has a number to act on — and it was computed in the same $\Theta(n\lg n)$ pass that produced the schedule, at no extra asymptotic cost.

3.17 Common Mistakes, and How to Avoid Them

1. Using O when you mean Θ. "Merge Sort is $O(n^2)$" is a true statement and a useless one. O is an upper bound, and any looser function is also an upper bound. *Fix:* state the tightest bound you can prove, and say Θ when you have both directions.

2. Reading an asymptotic equation backward. $2n = O(n^2)$ is fine; $O(n^2) = 2n$ is nonsense. *Fix:* read $=$ here as "is a member of", always left to right.

3. Leaving a positive leftover in a substitution proof. Reaching $T(n) \leq cn^2 + n$ and concluding "$cn^2 + n$ is $O(n^2)$, done" is the single most common error in this chapter (§3.11.2). The induction must reproduce the hypothesis *exactly*. *Fix:* subtract a lower-order term from the guess and try again (§3.11.3).

4. Forgetting the base case. An inductive step that works for all $n \geq 2$ proves nothing if the claim fails at the smallest n. *Fix:* check the base explicitly; if it fails at $n = 1$, move the base up to $n = 2$ and $n = 3$, which asymptotic notation permits (§3.11.4).

5. Mistaking "larger" for "polynomially larger". This is the Master Theorem's classic trap. $n\lg n$ is larger than n, but not larger by a factor of n^{ε} for any $\varepsilon > 0$, so $T(n) = 2T(n/2) + n\lg n$ is **not**

Case 3. *Fix:* compare exponents, not functions. If the only difference is a logarithmic factor, you are in a gap.

6. Skipping the regularity condition in Case 3. It is automatic for polynomial f, which lulls people into never checking it. It is *not* automatic in general (§3.13.4, §3.14.20). *Fix:* write the check down every time; it costs two lines.

7. Applying the Master Theorem to a recurrence that is not of its form. $T(n) = T(n/3) + T(2n/3) + n$ has unequal subproblems; $T(n) = T(n-1) + n$ subtracts instead of dividing; $T(n) = 2^n T(n/2) + n^n$ has a non-constant a. None of them is $aT(n/b) + f(n)$ with constant $a \geq 1$ and $b > 1$. *Fix:* check the form before reaching for the recipe.

8. Confusing $\lg^2 n$ with $\lg\lg n$. $\lg^2 n = (\lg n)^2$; $\lg\lg n$ is the logarithm of the logarithm. At $n = 65536$ the first is 256 and the second is 4. *Fix:* read the superscript as a power on the whole $\lg n$.

9. Worrying about the base of a logarithm inside Θ. $\Theta(\log_2 n) = \Theta(\log_{10} n) = \Theta(\ln n)$, because change of base multiplies by a constant. But in $n^{\log_b a}$ the base is decisive. *Fix:* bases do not matter inside the asymptotic notation and always matter in an exponent.

10. Assuming every two functions are comparable. Trichotomy fails (§3.7.6). *Fix:* remember n against $n^{1+\sin n}$.

11. Believing asymptotics predict small-input behavior. They do not. Insertion Sort beats Merge Sort below about twenty elements. *Fix:* use asymptotics to choose the algorithm and measurement to choose the cut-off.

12. Dropping a constant that was not a constant. In $T(n) = aT(n/b) + f(n)$, a and b must not depend on n. *Fix:* if the branching factor grows with n, you have a different kind of recurrence.

3.18 Chapter Summary

1. **Correctness first.** An algorithm is correct if it terminates and gives the right answer for *every* legal input. Partial correctness plus termination equals total correctness. Loop invariants — initialization, maintenance, termination — are induction applied to loops.
2. **Five notations.** O is an upper bound, Ω a lower bound, Θ a tight bound, o strictly smaller, ω strictly larger. Each is a statement about constants c (or c_1, c_2) and a threshold n_0 you can write down. To prove a bound, exhibit the constants; to disprove one, show every choice fails.
3. **The rules of the algebra.** All five notations are transitive; O, Ω, Θ are reflexive; only Θ is symmetric; O and Ω are transpose-symmetric. Sums take the maximum, products multiply, constants vanish. Every polynomial is Θ of its leading power, which is *why* we drop constants and lower-order terms.
4. **The common functions,** in order: 1, $\lg^* n$, $\lg\lg n$, $\lg n$, $\sqrt{n}$, n, $n\lg n$, n^2, n^3, 2^n, $n!$, n^n. Logarithms lose to every positive power of n; polynomials lose to every exponential; $\lg(n!) = \Theta(n\lg n)$.
5. **The summations.** Arithmetic $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$, by Gauss's pairing. Geometric $\sum_{i=0}^{n} a^i = \frac{1-a^{n+1}}{1-a}$, dominated by its first term when $a < 1$, its last when $a > 1$, and equal to $n + 1$ when $a = 1$. Harmonic $H_n = \Theta(\lg n)$. Telescoping collapses. When all else fails, bound by an integral.
6. **Three methods for recurrences.** Substitution proves; recursion trees suggest; the master method dispatches. Use the tree to guess, substitution to prove, the Master Theorem whenever the form fits.

7. **The Master Theorem.** For $T(n) = aT(n/b) + f(n)$, compare $f(n)$ against the leaf count $n^{\log_b a}$. Polynomially smaller $\Rightarrow \Theta(n^{\log_b a})$; equal $\Rightarrow \Theta(n^{\log_b a}\lg n)$; polynomially larger *and* regular $\Rightarrow \Theta(f(n))$. Two gaps exist, and the theorem is silent inside them.
8. **Merge Sort.** Divide at the midpoint, conquer recursively, combine by merging. $T(n) = 2T(n/2) + \Theta(n) = \Theta(n\lg n)$ in every case, proved three ways. Correct by loop invariant, stable, but not in place.

Table 3.9 is the one-page reference for the whole chapter, and **Table 3.10** ties the algorithms of Chapters 1–3 to their recurrences.

Table 3.9 Everything in Chapter 3, on one page.

Topic	Key tool	Purpose	When to use	Example
Correctness	loop invariants	prove the algorithm works	any iterative algorithm	Insertion Sort, MERGE
Upper bounds	Big-O	worst-case guarantee	bounding a running time	$7n - 3 = O(n)$
Lower bounds	Big-Ω	best case, problem bounds	showing a limit	unsorted search is $\Omega(n)$
Tight bounds	Big-Θ	exact growth rate	precise analysis	$\frac{n(n+1)}{2} = \Theta(n^2)$
Recurrence (prove)	substitution	rigorous proof	you already have a guess	$4T(n/2) + n$
Recurrence (explore)	recursion tree	visual intuition	new recurrence, need a guess	$T(n/3) + T(2n/3) + n$
Recurrence (instant)	Master Theorem	immediate answer	form $aT(n/b) + f(n)$	Merge Sort, binary search

Table 3.10 The algorithms met so far, their recurrences, and how each is solved.

Algorithm	Recurrence	Solution
Merge Sort	$T(n) = 2T(n/2) + \Theta(n)$	Master Case 2: $\Theta(n\lg n)$
Binary search	$T(n) = T(n/2) + \Theta(1)$	Master Case 2: $\Theta(\lg n)$
Insertion Sort	not recursive — use a summation	$\sum_{j=2}^{n}(j-1) = \Theta(n^2)$ worst case
Naive matrix multiply (D&C)	$T(n) = 8T(n/2) + \Theta(n^2)$	Master Case 1: $\Theta(n^3)$
Strassen	$T(n) = 7T(n/2) + \Theta(n^2)$	Master Case 1: $\Theta(n^{\lg 7})$
Recursive Fibonacci (Chapter 1)	$T(n) = T(n-1) + T(n-2) + \Theta(1)$	not a master form: $\Theta(\phi^n)$ — memoize instead

3.18.1 Looking ahead

Chapter 4, *Evaluation and Discussion I*, sets problems on Chapters 1–3 and works every one in full. Chapter 5 returns to sorting with **Heapsort**, which achieves $\Theta(n\lg n)$ *in place*, and **Quicksort**, whose worst case is $\Theta(n^2)$ but whose average case is $\Theta(n\lg n)$ with constants so small it usually wins in practice. Both analyses are recurrences, and you now have everything needed to solve them.

3.19 Exercises

3.1 Prove from the definition that $10n + 5 = O(n)$. Give explicit c and n_0, then find the smallest integer c that works with $n_0 = 1$.

3.2 Prove that $3n^2 + 2n = \Theta(n^2)$, giving c_1, c_2 and n_0.

3.3 Prove that $2^n \neq O(n^2)$.

3.4 Rank by growth, slowest first: $\lg^2 n$, $n^{1/3}$, $2^{\lg n}$, $n\lg n$, $4^{\lg n}$, n^3, 2^n, $n!$.

3.5 True or false, with a one-line justification each: (a) $n^2 = O(n^3)$; (b) $n^3 = O(n^2)$; (c) $2^{n+1} = O(2^n)$; (d) $2^{2n} = O(2^n)$; (e) $\lg(n^2) = \Theta(\lg n)$.

3.6 Prove $\sum_{i=1}^{n}(2i-1) = n^2$ twice: once with the arithmetic-series formula, once by induction.

3.7 Evaluate $\sum_{i=0}^{n} 3^i$ in closed form and give its Θ class.

3.8 Solve $T(n) = 2T(n/2) + n^3$ with the Master Theorem, including the regularity check.

3.9 Solve $T(n) = T(9n/10) + n$.

3.10 Solve $T(n) = 16T(n/4) + n$.

3.11 Solve $T(n) = 7T(n/3) + n^2$, including regularity.

3.12 Solve $T(n) = 2T(n/4) + n^{0.51}$, including regularity.

3.13 Solve $T(n) = T(n/2) + n/\lg n$. (Careful: this is *not* the gap case of §3.14.16.)

3.14 Solve $T(n) = T(n-1) + n$ with $T(1) = 1$. Why can the Master Theorem not be used?

3.15 Use a recursion tree to solve $T(n) = 3T(n/2) + n$, then confirm with the Master Theorem.

3.16 Prove $T(n) = 2T(n/2) + 1$ is $O(n)$ by substitution. The obvious guess fails; strengthen it.

3.17 Merging two sorted lists of m elements each: what is the smallest number of comparisons the merge of **Listing 3.8** can perform, and the largest? Give an input achieving each.

3.20 Solutions to the Exercises

3.1 For $n \geq 1$ we have $5 \leq 5n$, so $10n + 5 \leq 10n + 5n = 15n$. Take $c = 15$, $n_0 = 1$. For the smallest c with $n_0 = 1$: at $n = 1$ the left side is 15 and we need $15 \leq c \cdot 1$, so $c \geq 15$; and $c = 15$ works for all $n \geq 1$ as shown. **Smallest c is 15.** (With a larger threshold a smaller c works: $c = 11$ needs $5 \leq n$, so $n_0 = 5$.)

3.2 *Upper:* for $n \geq 1$, $2n \leq 2n^2$, so $3n^2 + 2n \leq 5n^2$; take $c_2 = 5$. *Lower:* $3n^2 + 2n \geq 3n^2$ for all $n \geq 0$; take $c_1 = 3$. Both hold from $n_0 = 1$. Hence $3n^2 + 2n = \Theta(n^2)$ with $c_1 = 3$, $c_2 = 5$, $n_0 = 1$.

3.3 Suppose $2^n \leq cn^2$ for all $n \geq n_0$. Then $\frac{2^n}{n^2} \leq c$ for all such n. But $\frac{2^n}{n^2} \to \infty$ by §3.8.5 (every exponential beats every polynomial), so it eventually exceeds c — a contradiction. Concretely, at $n = 30$, $2^{30} = 1073741824$ while $30^2 = 900$, so the ratio already exceeds 1.19×10^6. Hence $2^n \neq O(n^2)$. ■

3.4 First simplify: $2^{\lg n} = n$, and $4^{\lg n} = (2^2)^{\lg n} = 2^{2\lg n} = n^2$. Ranking, slowest first:

$$\lg^2 n \;<\; n^{1/3} \;<\; 2^{\lg n} = n \;<\; n\lg n \;<\; 4^{\lg n} = n^2 \;<\; n^3 \;<\; 2^n \;<\; n!.$$

The first inequality holds because $\lg^2 n = o(n^\varepsilon)$ for every $\varepsilon > 0$, in particular $\varepsilon = 1/3$.

3.5 (a) **True**: $n^2 \leq n^3$ for $n \geq 1$, so $c = 1$, $n_0 = 1$. (b) **False**: $n^3/n^2 = n \to \infty$, so no constant bounds it (same argument as §3.3.3). (c) **True**: $2^{n+1} = 2 \cdot 2^n$, so $c = 2$, $n_0 = 1$. (d) **False**: $2^{2n} = (2^n)^2$, so $2^{2n}/2^n = 2^n \to \infty$. (e) **True**: $\lg(n^2) = 2\lg n$, and a constant factor is absorbed by Θ.

3.6 *By formula:* $\sum_{i=1}^{n}(2i-1) = 2\sum_{i=1}^{n} i - \sum_{i=1}^{n} 1 = 2 \cdot \frac{n(n+1)}{2} - n = n^2 + n - n = n^2$. *By induction:* Basis $n = 1$: the sum is $2(1) - 1 = 1 = 1^2$. ✓ Step: assume $\sum_{i=1}^{n-1}(2i-1) = (n-1)^2$. Then $\sum_{i=1}^{n}(2i-1) = (n-1)^2 + (2n-1) = n^2 - 2n + 1 + 2n - 1 = n^2$. ✓ ■

3.7 Geometric with $a = 3 \neq 1$: $\sum_{i=0}^{n} 3^i = \frac{1-3^{n+1}}{1-3} = \frac{3^{n+1}-1}{2}$. Since $a > 1$ the last term dominates, so this is $\Theta(3^n)$. Check at $n = 3$: $1 + 3 + 9 + 27 = 40$ and $\frac{81-1}{2} = 40$. ✓

3.8 $a = 2$, $b = 2$, $f(n) = n^3$. $\log_2 2 = 1$, so $n^{\log_b a} = n$. Exponent $3 > 1$; take $\varepsilon = 2$, giving $f(n) = \Omega(n^{1+2})$. ✓ Regularity: $2f(n/2) = 2(n/2)^3 = 2 \cdot \frac{n^3}{8} = \frac{n^3}{4} = \frac{1}{4}f(n)$, so $c = 1/4 < 1$. ✓ **Case 3:** $T(n) = \Theta(n^3)$**.**

3.9 $a = 1$, $b = \frac{10}{9}$, $f(n) = n$. $\log_{10/9} 1 = 0$, so $n^{\log_b a} = 1$. Exponent $1 > 0$; take $\varepsilon = 1$. ✓ Regularity: $1 \cdot f(9n/10) = \frac{9n}{10} = 0.9\, f(n)$, so $c = 0.9 < 1$. ✓ **Case 3:** $T(n) = \Theta(n)$**.** (Shaving a mere tenth off the input each time still gives linear total work — the geometric series $n(1 + 0.9 + 0.81 + \cdots) = 10n$.)

3.10 $a = 16$, $b = 4$, $f(n) = n$. $\log_4 16 = 2$, so $n^{\log_b a} = n^2$. Exponent $1 < 2$; take $\varepsilon = 1$, giving $f(n) = O(n^{2-1})$. ✓ **Case 1:** $T(n) = \Theta(n^2)$**.**

3.11 $a = 7$, $b = 3$, $f(n) = n^2$. $\log_3 7 = \frac{\ln 7}{\ln 3} = \frac{1.9459}{1.0986} \approx 1.771$. Exponent $2 > 1.771$; take $\varepsilon = 0.2$, target $n^{1.971}$, and $n^2 = \Omega(n^{1.971})$. ✓ Regularity: $7(n/3)^2 = \frac{7}{9}n^2 \approx 0.778\, f(n)$, so $c = 7/9 < 1$. ✓ **Case 3:** $T(n) = \Theta(n^2)$**.**

3.12 $a = 2$, $b = 4$, $f(n) = n^{0.51}$. $\log_4 2 = 0.5$, so $n^{\log_b a} = n^{0.5}$. Exponent $0.51 > 0.5$; take $\varepsilon = 0.01$. ✓ Regularity: $2(n/4)^{0.51} = \frac{2}{4^{0.51}} n^{0.51}$, and $4^{0.51} = e^{0.51 \ln 4} = e^{0.7070} \approx 2.028$, so $c \approx \frac{2}{2.028} \approx 0.986 < 1$. ✓ **Case 3:** $T(n) = \Theta(n^{0.51})$**.** Note how narrow the margin is — ε may be as small as you like, but it must be a fixed positive constant.

3.13 $a = 1$, $b = 2$, $f(n) = n/\lg n$. $\log_2 1 = 0$, so $n^{\log_b a} = 1$. Is $f(n) = \Omega(n^{0+\varepsilon})$? With $\varepsilon = 1/2$: $\frac{n/\lg n}{n^{0.5}} = \frac{n^{0.5}}{\lg n} \to \infty$. ✓ Regularity: $f(n/2) = \frac{n/2}{\lg n - 1}$, and $\frac{f(n/2)}{f(n)} = \frac{n/2}{\lg n - 1} \cdot \frac{\lg n}{n} = \frac{\lg n}{2(\lg n - 1)} \to 1/2$; for $n \geq 16$ it is at most $2/3$. ✓ **Case 3:** $T(n) = \Theta(n/\lg n)$**.** The contrast with §3.14.16 is the point: there $a = 2$ made the leaf count n, which $n/\lg n$ could not beat; here $a = 1$ makes the leaf count 1, which it beats easily.

3.14 Not a master form: the argument is $n - 1$, not n/b, so there is no constant $b > 1$. Solve by telescoping. Summing $T(i) - T(i-1) = i$ for $i = 2, \ldots, n$ gives $T(n) - T(1) = \sum_{i=2}^{n} i$, so $T(n) = 1 + \left(\frac{n(n+1)}{2} - 1\right) = \frac{n(n+1)}{2} = \Theta(n^2)$.

3.15 *Tree.* Level 0 costs n. Level 1 has 3 nodes of size $n/2$, costing $3 \cdot \frac{n}{2} = \frac{3}{2}n$. Level 2 has 9 nodes of size $n/4$, costing $9 \cdot \frac{n}{4} = \frac{9}{4}n$. The level ratio is $3/2 > 1$, so the series is **increasing** and the leaves dominate. The tree has $\lg n$ levels and $3^{\lg n} = n^{\lg 3}$ leaves, so

$$T(n) = \Theta(n^{\lg 3}) \approx \Theta(n^{1.585}).$$

Master. $a = 3$, $b = 2$, $\log_2 3 \approx 1.585$; $f(n) = n$ has exponent $1 < 1.585$, so with $\varepsilon = 0.5$ we get Case 1 and the same answer. ✓

3.16 The obvious guess $T(n) \le cn$ fails: $2 \cdot c\frac{n}{2} + 1 = cn + 1 > cn$. Strengthen to $T(n) \le cn - d$ with $d > 0$:

$$\begin{aligned} T(n) = 2T(n/2) + 1 &\le 2\left(c\frac{n}{2} - d\right) + 1 = cn - 2d + 1 \\ &= (cn - d) - (d - 1) \le cn - d \quad \text{whenever } d \ge 1. \end{aligned}$$

Base case $n = 1$ with $T(1) = 1$: need $1 \le c - d$, so $c \ge 1 + d$; with $d = 1$, take $c = 2$. Hence $T(n) \le 2n - 1 = O(n)$. (Exactly right, in fact: $T(n) = 2n - 1$.)

3.17 *Smallest:* m comparisons. If every element of the left list is $\le$ every element of the right list — for example $L = [1,2,3]$, $R = [4,5,6]$ — the loop takes all m left elements, using one comparison each, and then exits because L is exhausted; the remaining right elements are copied by the clean-up loop without comparison. *Largest:* $2m - 1$ comparisons. Each iteration of the loop makes one comparison and copies one element, and the loop stops as soon as either list is empty. When it stops, the other list still holds at least one element, so the loop has copied at most $2m - 1$ of the $2m$ elements and made at most $2m - 1$ comparisons. The maximum is reached when both lists stay non-empty until the very end, which happens when their elements interleave. An input achieving it: $L = [1,3,5]$, $R = [2,4,6]$. The comparisons are 1 vs 2, 3 vs 2, 3 vs 4, 5 vs 4, 5 vs 6 — five of them, which is $2 \cdot 3 - 1$ — and then 6 is copied without a comparison. *Consistency check.* For $n = 8$ the whole sort does 4 merges of $1 + 1$ (one comparison each), 2 merges of $2 + 2$ (at most 3 each), and 1 merge of $4 + 4$ (at most 7): at most $4 + 6 + 7 = 17$. **Listing 3.8** reported 16 for [1,5,2,4,6,3,2,6], one below the maximum — consistent. ✓

Further reading

The formal statement of the Master Theorem, the regularity condition, and the oscillating counterexample of §3.14.20 follow [1, ch. 4]; the loop-invariant treatment of MERGE follows [1, ch. 2]. The summation identities and the integral-bounding technique of §3.9.6 are treated at length in [2]. The framing of divide and conquer as a design strategy, and the inversion-counting application of §3.16.1, follow [3, ch. 5]. Practical cut-offs between Merge Sort and Insertion Sort, and the stability argument of §3.15.7, are discussed in [4] and analyzed exhaustively in [5]. The competitive-programming perspective on when an $O(n^2)$ method is still acceptable is [6].

References

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

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

[3] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[4] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

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

[6] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, 4th ed. Raleigh, NC, USA: Lulu Press, 2020.

Chapter 4

Evaluation and Discussion I: Foundations, Analysis, and First Problems

This chapter pauses the forward march of the book and consolidates everything Chapters 1 to 3 have built. It contains no new theory. Instead it works problems — fourteen of them, every one solved from nothing, with every intermediate number written down. By the end you will have a seven-step method for reading any algorithms problem, and you will have seen it carry two real online-judge problems, LDCSUMAB and TOHU, from a blank page to an accepted program, including a proof that the program is correct and a measurement of what it costs. You will have proved a loop invariant line by line, traced Insertion Sort and binary search in tables, counted the exact number of steps a triple loop takes, proved and disproved asymptotic claims with explicit constants c and n_0, ranked thirteen functions by growth, and solved recurrences by substitution, by a recursion tree and by the Master Theorem — including one recurrence where the Master Theorem refuses to answer and you must finish the job yourself. Finally you will have timed a brute-force algorithm against a better one on a real machine and watched the difference between $\Theta(n^2)$ and $\Theta(n\lg n)$ grow from a rounding error into a factor of more than a hundred. Every program in the chapter is complete, compiles with `g++ -std=c++17 -O2 -Wall`, and is quoted with the output it really printed.

4.1 Where We Stand

Three chapters have given us a toolkit. It is worth listing what is in it before we start using it, because every problem in this chapter will reach for one of these tools, and naming the tool is half of the work.

From **Chapter 1** we have the idea that an algorithm is a precise, finite procedure that turns an input into an output; the two goals of correctness and efficiency; the five design paradigms (incremental, greedy, divide and conquer, dynamic programming, mathematical); and the five-phase framework *Understand, Abstract, Design, Analyze, Implement* for attacking a problem.

From **Chapter 2** we have the RAM model, in which each simple operation costs one step; the loop invariant, with its three obligations of initialization, maintenance and termination; Insertion Sort and its best-case, worst-case and average-case running times; linear search; and binary search with its $O(\lg n)$ bound.

From **Chapter 3** we have the asymptotic notations O, Ω, Θ, o and ω, each defined by explicit constants; the standard functions (floors, ceilings, logarithms, exponentials, factorials, Fibonacci numbers); the summation toolkit (arithmetic, geometric and harmonic series, telescoping sums, bounds by integrals); and the three methods for recurrences — substitution, the recursion tree, and the Master Theorem. All of it follows the treatment of Cormen et al. [1], and the discrete mathematics behind the summations is the standard material of Rosen [2].

This chapter uses every one of those tools on real problems. It is organized in three layers.

1. **A method** (Section 4.2). Seven steps that turn "read the problem carefully" into a procedure you can follow with a pencil.
2. **Two problems from the SPOJ online judge** (Sections 4.3 and 4.4), each worked in full detail: the statement restated in plain words, the seven steps applied one at a time, the first cases computed by hand, the formula derived, the formula proved correct, the running time analyzed, a complete C++17 program compiled and run, a second independent solution, and the mistakes people typically make. These two walkthroughs are deliberately slow. If you have never solved an online-judge problem before, you should be able to reproduce them keystroke for keystroke.
3. **Twelve worked problems** (Sections 4.5 to 4.9), each short and each aimed at one tool from Chapters 1 to 3. Every one has the same shape: the statement, what it tests, the solution with every arithmetic step, the answer, a program where a program helps, and a *check yourself* variant with its answer so that you can confirm you have really understood.

Section 4.10 then lays all fourteen problems side by side, Section 4.11 shows where the same ideas appear in a real product, and the chapter closes with the common mistakes, a summary, twelve exercises and their full solutions.

> **HOW TO USE THIS CHAPTER.** Read each problem statement, then cover the solution with a sheet of paper and try it yourself for ten minutes before reading on. A solution you have struggled with for ten minutes is understood in a way that a solution you have only read never is. The *check yourself* variants exist so that you can test that understanding on something you have not seen worked.

4.2 How to Read an Algorithms Problem

This section is a method, not advice. Advice is "read the problem carefully". A method tells you what to write on the page, in what order, before you allow yourself to think about a solution. Do these seven steps in order, every time. On an easy problem they take four minutes. On a hard one they are the difference between solving it and staring at it.

The seven steps are the five phases of Chapter 1 made concrete. Steps 1 and 2 are *Understand*; Steps 3, 4 and 5 are *Abstract*; Step 6 is *Design* together with the correctness half of *Analyze*; Step 7 is *Implement*, preceded by the cost half of *Analyze*. Sections 4.3 and 4.4 apply them one at a time, with a heading for each, so that you can see the method working rather than merely read about it.

4.2.1 Step 1 — Copy the input and output formats, verbatim

Before anything else, copy onto your page, word for word:

- the **input format** — how many lines, what is on each, in what order;
- the **output format** — how many lines, what precision, what exact wording;
- the **constraints** — every bound on every variable;
- the **sample input and sample output**, character for character.

Copy them even though they are on the screen in front of you. The act of copying forces you to read every token. A large share of all wrong-answer verdicts on an online judge are format errors, not algorithm errors: a missing newline, six decimal places where eleven were asked for, two numbers printed in the wrong order. In the TOHU problem of Section 4.4 the output must have exactly eleven digits after the decimal point; print ten and a perfect algorithm is judged wrong.

4.2.2 Step 2 — Name the unknown, and name the data

Write two short lists.

Given: the things the input supplies, each with a symbol. *Wanted:* the thing the output demands, as a formula or a sentence with a symbol in it.

For the LDCSUMAB problem of Section 4.3 this reads:

- Given: a position k, with $1 \leq k \leq 10^9$, up to $T \leq 10$ times.
- Wanted: the exponents $m < n$ such that $2^m + 2^n$ is the k-th smallest number of that form.

Now you have symbols to compute with. Until you have symbols you cannot do algebra, and algebra is where most of the problems in this chapter are solved.

4.2.3 Step 3 — Turn the constraints into a time budget

This is the step that decides which algorithms are allowed, and it takes ten seconds.

A modern judge machine executes roughly 10^8 to 10^9 simple operations per second; the rules of thumb below follow standard competitive-programming practice [3]. Time limits on the SPOJ judge [4] are usually one to three seconds. So the rule of thumb is: **your program may perform about 10^8 operations in total.** Multiply the number of test cases by the work per test case, and compare with 10^8.

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
$n \leq 10$	$O(n!)$, $O(2^n \cdot n)$	brute force, permutations
$n \leq 20$	$O(2^n)$	subset enumeration, bitmask DP
$n \leq 500$	$O(n^3)$	Floyd–Warshall, matrix chain
$n \leq 5000$	$O(n^2)$	pairwise DP, simple DP tables
$n \leq 10^6$	$O(n)$ or $O(n\log n)$	one pass, sorting, sieve, stack
$n \leq 10^{12}$	$O(\sqrt{n})$	divisor blocks, trial division
$n \leq 10^{18}$	$O(\log n)$	binary search, fast exponentiation
n unbounded	$O(1)$	a closed form

Use **Table 4.1** ruthlessly. It is the RAM model of Chapter 2 turned into a planning tool: count the steps, compare with the budget, and throw away every plan that does not fit. Sometimes a problem states its target complexity outright — the TOHU problem is usually posed with the note *time complexity* $O(1)$ — and then it is telling you the shape of the answer: $O(1)$ means there is a closed form and your job is to find it.

4.2.4 Step 4 — Compute the first five cases by hand

Not by program. By hand, on paper, in a table, with the arithmetic shown. This is the step beginners skip, and it is the step that solves the problem.

For TOHU the table of the first four values of the unknown sequence reads $2/3, \frac{1}{24}, \frac{1}{60}, \frac{1}{120}$, and anyone who has met Chapter 3's telescoping sums recognizes the last three as $\frac{1}{2\cdot3\cdot4}, \frac{1}{3\cdot4\cdot5}, \frac{1}{4\cdot5\cdot6}$. For LDCSUMAB the table of the first ten answers shows $n = 1,2,2,3,3,3,4,4,4,4$ — one 1, two 2s, three

3s, four 4s — and the triangular numbers of Chapter 3 jump off the page. No insight arrives from the sky. It comes out of a table of ten numbers.

> **THE TABLE IS THE ALGORITHM.** In most of the problems in this chapter the solution is visible in a hand-computed table of the first five to ten cases. Build the table before you theorize. If the table shows a pattern, prove it. If it shows no pattern, you have learned something too: the problem is structural rather than numerical, and you should be looking at invariants, orderings or exchange arguments instead.

4.2.5 Step 5 — Name the shape

Say out loud which family the problem belongs to. There are not many families, and the problems of this chapter use only six of them:

1. **Closed form from a sum** — arithmetic series, telescoping, partial fractions. (TOHU; Laundry Tokens in Section 4.8.)
2. **Inverting a counting formula** — find where a position falls when you know how many items each block holds. (LDCSUMAB.)
3. **A loop with an invariant** — a running quantity that means the same thing before every iteration. (Horner's rule, Insertion Sort, binary search, the Profitina Laundry case study.)
4. **Counting steps with sums** — nested loops become nested sums, and nested sums become closed forms. (The triple loop of Section 4.6.)
5. **A recurrence** — divide-and-conquer recurrences solved by substitution, recursion tree or Master Theorem, and linear recurrences such as Fibonacci. (Sections 4.7 and 4.8.)
6. **Brute force, then better** — the obvious algorithm is correct and too slow; a sort or a search removes a factor of n. (Section 4.9.)

If you can put the problem in a family within five minutes, you are most of the way home, because the family comes with a method attached.

4.2.6 Step 6 — Write the invariant, the recurrence, or the formula, and prove it

Before code. One line, in mathematics, that says what is true — and then the argument that it *is* true. Examples from this chapter:

- LDCSUMAB: the answer's exponent n is the smallest integer with $\frac{n(n+1)}{2} \geq k$.
- TOHU: $S_n = \frac{3}{4} - \frac{1}{2(n+1)(n+2)}$.
- Horner's rule: *at the start of each iteration, p holds the value of the polynomial formed by the coefficients already processed.*
- The $2 \times N$ strip: $D_N = D_{N-1} + D_{N-2}$, with $D_0 = D_1 = 1$.

A pattern read off a table is a conjecture; this step turns it into a theorem. Chapter 2's loop invariants and Chapter 3's induction and substitution are the tools, and every problem below uses one of them.

4.2.7 Step 7 — Only now, write the program

And when you write it, write it with the constraints in your hand. Three checks before you submit:

1. **Does anything overflow?** A 32-bit `int` holds values up to $2^{31} - 1 \approx 2.1 \times 10^9$; a 64-bit `long long` holds up to $2^{63} - 1 \approx 9.22 \times 10^{18}$. Estimate the largest intermediate value — not just the

answer — before choosing the type. In TOHU, $2(n+1)(n+2)$ overflows an `int` long before n reaches a million.

2. **Does the precision hold?** A `double` carries about fifteen to sixteen significant decimal digits; a `float` carries about seven. TOHU asks for eleven digits after the decimal point of a number below 1, so `double` survives and `float` does not.
3. **Does the input reader keep up?** With tens of thousands of test cases, `cin >>` without `ios::sync_with_stdio(false)` can be several times slower than `scanf`. Every program in this chapter uses `scanf` and `printf`, which are fast and whose format strings make the required precision explicit.

4.3 Problem 1 — LDCSUMAB: Sums of Two Powers of Two

The first of our two judge problems is LDCSUMAB, from the tutorial section of SPOJ [4]. It is the easier of the two, and it is chosen because it uses only three ideas from Chapters 2 and 3: the arithmetic series $1+2+\cdots+n$, the ceiling function, and binary search. **Figure 4.1** restates the problem page.

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 $0 \le m < n$, 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 ($T \le 10$), the number of test cases. Each of the next T lines contains one integer k ($k \le 10^9$).

Output

For each test case, print the two numbers m and n ($m < n$) of the k-th element.

Sample

Input

```
3
1
3
5
```

Output

```
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.

4.3.1 The problem in our own words

Take every number that can be written as the sum of two *different* powers of two, $2^m + 2^n$ with $0 \le m < n$. For instance $2^0 + 2^1 = 3$, $2^0 + 2^2 = 5$, $2^1 + 2^2 = 6$, $2^0 + 2^3 = 9$. Write all of these numbers in increasing order, forming an infinite list 3,5,6,9,10,12,17, ... You are given a position k in that list and must say which two exponents produced the number sitting there: print m and then n.

The input starts with the number of test cases T, at most 10. Each of the next T lines holds one position k, at most 10^9. For each one we print one line containing m and n, separated by a space. With the sample input `3`, `1`, `3`, `5` the expected output is `0 1`, `1 2`, `1 3`: the first number in the list is $3 = 2^0 + 2^1$, the third is $6 = 2^1 + 2^2$, and the fifth is $10 = 2^1 + 2^3$.

PROBLEM. List all integers of the form $2^m + 2^n$ with $0 \le m < n$ in increasing order. Given a position k ($1 \le k \le 10^9$), output the exponents m and n of the k-th number in the list. There are at most 10 queries.

4.3.2 Steps 1 to 3: the formats, the symbols, the budget

Step 1 — the formats, copied.

- Input: line 1 holds T ($T \le 10$). Then T lines, each with one integer k ($1 \le k \le 10^9$).
- Output: for each k, one line `m n` — the smaller exponent first.
- Sample input `3 / 1 / 3 / 5`; sample output `0 1 / 1 2 / 1 3`.

Notice the order of the output: m first, then n, and $m < n$. Printing them the other way round is a Wrong Answer on the very first sample line.

Step 2 — given and wanted.

- Given: k, a position in the sorted list, $1 \le k \le 10^9$.
- Wanted: the pair (m, n) with $0 \le m < n$ such that $2^m + 2^n$ is the k-th smallest number of that form.

Step 3 — the budget. Let us price the obvious ideas against **Table 4.1**.

- *Build the list.* Generate the numbers in order until we have k of them. That is $\Theta(k)$ work, up to 10^9 per query and 10^{10} for ten queries — a hundred times over the 10^8 budget, and it would need gigabytes of memory besides. Rejected.
- *Walk through the blocks.* We shall see in Step 4 that the list falls into blocks of sizes 1,2,3, ... Stepping from block to block until we reach position k takes about $\sqrt{2k}$ steps, roughly 44,721 for $k = 10^9$, so about 4.5×10^5 for ten queries. That fits.
- *Binary search over the blocks.* About $\lg 50{,}000 \approx 16$ steps per query. Fits easily.
- *A closed formula.* $O(1)$ per query. The best possible.

So even the second idea is fast enough; the budget does not *force* a formula here. We will derive the formula anyway, because deriving it teaches the most, and then present binary search as the alternative in Section 4.3.8. This is a common situation: the budget tells you which ideas are *forbidden*, not which one is best.

4.3.3 Step 4: the first cases by hand

We need to see the list. The only honest way to be sure of its order is to write out every candidate and sort. With $n \le 4$ there are ten candidates:

- $n = 1$: only $m = 0$, giving $2^0 + 2^1 = 1 + 2 = 3$.
- $n = 2$: $m = 0$ gives $1 + 4 = 5$; $m = 1$ gives $2 + 4 = 6$.
- $n = 3$: $m = 0,1,2$ give $1 + 8 = 9$, $2 + 8 = 10$, $4 + 8 = 12$.
- $n = 4$: $m = 0,1,2,3$ give $1 + 16 = 17$, $2 + 16 = 18$, $4 + 16 = 20$, $8 + 16 = 24$.

The smallest candidate with $n = 5$ is $1 + 32 = 33$, larger than all ten above, so these ten are exactly the first ten numbers of the list. Sorting them gives **Table 4.2**.

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, ...

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

Read the table column by column, as Step 4 asks.

- The n **column** is 1,2,2,3,3,3,4,4,4,4. Block n holds exactly n entries.
- The m **column**, inside each block, runs $0,1,\ldots,n-1$ in order.
- The **last position of each block** is 1,3,6,10. These are the triangular numbers of Chapter 3.

Figure 4.2 shows the same observation drawn as three tables side by side: the list, the map from k to n, and the inversion we are about to derive.

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).

4.3.4 Steps 5 and 6: the shape, and the formula

Step 5 — the shape. This is the family *inverting a counting formula*. We know how many entries each block holds; we must find which block contains position k, and where inside that block k falls. Problems of this kind are called **ranking and unranking**: *ranking* turns an object into its position, *unranking* turns a position back into the object.

Step 6 — the formula. A table of ten rows is evidence, not proof. Before building on the pattern we must check that it continues forever. Two claims settle it.

Claim 1: every value with larger exponent $n+1$ exceeds every value with exponent n. The largest value in block n uses the largest allowed m, which is $n-1$:

$$2^{n-1} + 2^n = 2^{n-1} + 2 \cdot 2^{n-1} = 3 \cdot 2^{n-1}.$$

The smallest value in block $n+1$ uses $m = 0$:

$$2^0 + 2^{n+1} = 1 + 4 \cdot 2^{n-1}.$$

Since $3 \cdot 2^{n-1} < 4 \cdot 2^{n-1} < 1 + 4 \cdot 2^{n-1}$, every value of block n is smaller than every value of block $n+1$. The blocks never interleave.

Claim 2: inside block n, the values increase with m. For fixed n, raising m by one raises 2^m and leaves 2^n alone, so $2^m + 2^n$ grows.

Together the two claims say that the list is exactly the pairs (m, n) in **lexicographic order by n first, then by m**:

$$(0,1),\ (0,2), (1,2),\ (0,3), (1,3), (2,3),\ (0,4), (1,4), (2,4), (3,4),\ \ldots$$

Counting the blocks. Block n holds the pairs with $m = 0,1, \ldots, n-1$, so it has exactly n entries. The number of entries in blocks 1 through n together is therefore the arithmetic series of Section 3.9.1,

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \tag{4.1}$$

the n-th **triangular number**, which we shall write T_n. Recall why Equation (4.1) holds: write the sum forwards and backwards, $S = 1 + 2 + \cdots + n$ and $S = n + (n-1) + \cdots + 1$, and add column by column; each of the n columns sums to $n+1$, so $2S = n(n+1)$. We will also need $T_0 = 0$ and the fact that $T_n = T_{n-1} + n$.

Which block holds position k? Blocks 1 to $n-1$ fill positions 1 to T_{n-1}; block n fills positions $T_{n-1} + 1$ to T_n. So position k lies in block n exactly when

$$T_{n-1} < k \leq T_n,$$

which is the same as saying: **n is the smallest integer with $T_n \geq k$.**

Solving for n. Treat $T_n = k$ as an equation in a real unknown n and solve it:

$$\frac{n(n+1)}{2} = k \ \Rightarrow\ n^2 + n = 2k \ \Rightarrow\ n^2 + n - 2k = 0.$$

This is a quadratic $an^2 + bn + c = 0$ with $a = 1, b = 1, c = -2k$. The quadratic formula $n = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ gives

$$n = \frac{-1 \pm \sqrt{1 - 4 \cdot 1 \cdot (-2k)}}{2} = \frac{-1 \pm \sqrt{1 + 8k}}{2}.$$

The minus sign gives a negative number, which cannot be a block index, so we keep the plus sign. Call the result $x = \frac{-1+\sqrt{1+8k}}{2}$. The function $T(n) = n(n+1)/2$ is increasing for $n \geq 0$, and $T(x) = k$ exactly. The smallest *integer* n with $T_n \geq k$ is therefore the smallest integer $n \geq x$, which is the ceiling of Section 3.8.2:

$$n = \left\lceil \frac{-1+\sqrt{1+8k}}{2} \right\rceil \tag{4.2}$$

Why the ceiling and not the floor? If k falls strictly inside a block, x is not an integer — for $k = 5$ it is about 2.70 — and position 5 lives in block 3, so we must round *up*. If k is exactly a triangular number — the last entry of its block — then x is an integer and the ceiling leaves it alone.

Where inside the block? Block n occupies positions $T_{n-1}+1, T_{n-1}+2, \ldots, T_n$, holding $m = 0,1,\ldots,n-1$ in that order (Claim 2). Position $T_{n-1}+1$ holds $m = 0$, position $T_{n-1}+2$ holds $m = 1$, and in general position k holds $m = k - T_{n-1} - 1$. Writing $r = T_n$ for the last position of the block, so that $T_{n-1} = r - n$,

$$m = k - (r - n) - 1 = n + k - (r + 1) \tag{4.3}$$

Working the cases by hand. Every step of Equation (4.2) and Equation (4.3), one line at a time.

$k = 1$. $1 + 8 \cdot 1 = 9$. $\sqrt{9} = 3$. $\frac{-1+3}{2} = 1$. $n = \lceil 1 \rceil = 1$. $r = \frac{1 \cdot 2}{2} = 1$. $m = 1 + 1 - (1 + 1) = 0$. Answer `0 1` ✓

$k = 3$. $1 + 24 = 25$. $\sqrt{25} = 5$. $\frac{-1+5}{2} = 2$. $n = 2$. $r = \frac{2 \cdot 3}{2} = 3$. $m = 2 + 3 - 4 = 1$. Answer `1 2` ✓

$k = 5$. $1 + 40 = 41$. $\sqrt{41} \approx 6.4031$. $\frac{-1+6.4031}{2} \approx 2.7016$. $n = \lceil 2.7016 \rceil = 3$. $r = \frac{3 \cdot 4}{2} = 6$. $m = 3 + 5 - 7 = 1$. Answer `1 3` ✓ — and **Table 4.2** confirms that position 5 holds $2^1 + 2^3 = 10$.

$k = 6$, *a block boundary.* $1 + 48 = 49$. $\sqrt{49} = 7$. $\frac{-1+7}{2} = 3$. $n = 3$. $r = 6$. $m = 3 + 6 - 7 = 2$. Answer `2 3`, the value $4 + 8 = 12$ ✓.

$k = 7$, *the first entry of a new block.* $1 + 56 = 57$. $\sqrt{57} \approx 7.5498$. $\frac{-1+7.5498}{2} \approx 3.2749$. $n = 4$. $r = \frac{4 \cdot 5}{2} = 10$. $m = 4 + 7 - 11 = 0$. Answer `0 4`, the value $1 + 16 = 17$ ✓.

$k = 10^9$, *the largest allowed.* $1 + 8 \times 10^9 = 8{,}000{,}000{,}001$. Its square root: $89{,}442^2 = 7{,}999{,}871{,}364$, leaving 128,637, and $128{,}637/(2 \times 89{,}442) \approx 0.719$, so $\sqrt{8{,}000{,}000{,}001} \approx 89{,}442.719$. Then $\frac{-1+89{,}442.719}{2} \approx 44{,}720.86$, so $n = 44{,}721$. Check it: $T_{44720} = 22{,}360 \times 44{,}721 = 999{,}961{,}560 < 10^9$ and $T_{44721} = 44{,}721 \times 22{,}361 = 1{,}000{,}006{,}281 \geq 10^9$. ✓ So $r = 1{,}000{,}006{,}281$ and $m = 44{,}721 + 1{,}000{,}000{,}000 - 1{,}000{,}006{,}282 = 38{,}439$. Answer `38439 44721`.

4.3.5 Proof of correctness

We have a formula and six hand checks. Now we prove that the formula is right for *every* k, so that no seventh check could ever fail.

> **THEOREM.** For every integer $k \geq 1$, let n be the smallest integer with $T_n \geq k$ and let $m = k - T_{n-1} - 1$. Then $0 \leq m < n$, and $2^m + 2^n$ is the k-th smallest integer of the form $2^a + 2^b$ with $0 \leq a < b$.

Proof. By Claims 1 and 2, the sorted list is the pairs in order of n, then of m. The pairs with exponent at most $n-1$ number T_{n-1} by Equation (4.1), and they occupy the first T_{n-1} positions; the pairs with exponent at most n occupy the first T_n positions. By the choice of n, $T_{n-1} < k$ (otherwise $n-1$ would be a smaller integer with $T_{n-1} \geq k$) and $k \leq T_n$. So position k is not among the first T_{n-1} positions but is among the first T_n: its exponent is exactly n. Inside block n the positions $T_{n-1}+1, \ldots, T_n$ carry $m = 0, \ldots, n-1$ in increasing order, so position k carries $m = k - T_{n-1} - 1$. Finally, $T_{n-1} < k$ gives $m \geq 0$, and $k \leq T_n = T_{n-1} + n$ gives $m \leq n-1$. ■

The program in Section 4.3.7 does not trust the floating-point square root blindly; it computes a first guess and then corrects it with two small loops. Their correctness is a small loop-invariant argument in the style of Chapter 2, and it is worth writing out because it is exactly what makes the program safe. **Pseudocode 4.1** is the procedure.

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 only stops when its test $T_n < k$ is false. It does stop, because T_n grows without bound as n increases.
- *Lines 4–5 keep the invariant* $T_n \geq k$. It holds on entry. The body runs only when $T_{n-1} \geq k$, and then it replaces n by $n-1$, whose triangular number is exactly that $T_{n-1} \geq k$. So the invariant survives every iteration. The loop terminates because n decreases and is bounded below by 0.
- *At exit* the test is false: either $n = 0$ or $T_{n-1} < k$. The first is impossible, since the invariant would give $T_0 = 0 \geq k \geq 1$. So $T_{n-1} < k \leq T_n$, and n is the smallest integer with $T_n \geq k$ — exactly the n of the Theorem.
- *Lines 6–7* compute $m = n + k - (r+1)$, which is Equation (4.3), which equals $k - T_{n-1} - 1$, which the Theorem proves correct.

4.3.6 Complexity

Each query costs one square root, a constant number of arithmetic operations, and the two correcting loops. How many times do the loops run? The honest answer, for $k \leq 10^9$, is *never*, and the reason is worth knowing.

At a block boundary $k = T_n = \frac{n(n+1)}{2}$ we have $1 + 8k = 1 + 4n(n+1) = 4n^2 + 4n + 1 = (2n+1)^2$ — always an exact odd perfect square. A `double` stores every integer up to $2^{53} \approx 9 \times 10^{15}$ exactly, and the IEEE-754 standard requires `sqrt` to be *correctly rounded*, which for an exactly representable exact answer means exactly right. With $k \leq 10^9$ we have $1 + 8k \leq 8 \times 10^9$, comfortably inside the exact range, so the first guess on line 1 is already correct and neither loop body executes. For inputs far beyond this problem's limits the guess could be off by one, and then each loop runs once. Either way the work per query is $O(1)$, and the whole program is $\Theta(T)$ time and $O(1)$ extra space.

Why keep the loops, then? Because the argument above depends on $k \leq 10^9$, and constraints change. If a later version of the problem allowed $k \leq 10^{18}$, then $1 + 8k \approx 8 \times 10^{18}$ exceeds 2^{53}, the exactness argument collapses, and a program without the loops would fail on some boundaries. Two `while` loops that almost never execute cost nothing and remove the dependence on an argument the reader has to reconstruct.

Overflow. The largest n we ever meet is 44,721 (plus one during correction), so $n(n+1) \approx 2.0 \times 10^9$. That is uncomfortably close to the `int` limit of 2,147,483,647, so the program uses `long long` throughout.

4.3.7 Step 7: the program

Figure 4.3 shows the table of Section 4.3.3 beside a compact C++ solution that implements it directly with `ceil`.

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

```cpp
#include <cmath>
#include <cstdio>

int main(){
    int T, k, n, r, m;
    scanf("%d",&T);
    while(T--){
        scanf("%d",&k);
        n=(int)ceil((-1.0+sqrt(1.0+8.0*(double)k))/2.0);
        r=n*(n+1)/2;
        m=n+k-(r+1);
        printf("%d %d\n",m,n);
    }
    return 0;
}
```

Figure 4.3 A compact LDCSUMAB solution beside the table it implements: `n = ceil((-1+sqrt(1+8k))/2)`, `r = n(n+1)/2`, `m = n + k - (r+1)`.

Listing 4.1 is the robust version of the same idea, line for line the procedure of **Pseudocode 4.1**. The floating-point square root is used only as a *starting guess*, and the two correction loops then nail down the exact integer.

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;
}
```

Compiling and running it, step by step. If you have never run a judge program before, do exactly this.

1. Save the listing in a file called `ldcsumab.cpp`.
2. Compile it: `g++ -std=c++17 -O2 -Wall -o ldcsumab ldcsumab.cpp`. The flag `-std=c++17` selects the language version, `-O2` turns on optimization as the judge does, and `-Wall` asks the compiler to warn about anything suspicious. The command should print nothing at all.
3. Save the sample input in a file `sample.txt`, one number per line: `3`, `1`, `3`, `5`.
4. Run the program with the file as its standard input: `./ldcsumab < sample.txt` (on Windows, `ldcsumab.exe < sample.txt`).

It prints

```
0 1
1 2
1 3
```

which is the sample output, character for character. With the positions 1 to 10 as input it prints

```
0 1
0 2
1 2
0 3
1 3
2 3
0 4
1 4
2 4
3 4
```

which is the (m, n) columns of **Table 4.2**, row for row. And with the positions $6, 7, 499500, 499501, 10^9$ — two boundaries, a triangular number $T_{999} = 499{,}500$, the entry after it, and the largest allowed k — it prints

```
2 3
0 4
998 999
0 1000
38439 44721
```

Each line agrees with a hand calculation: position 499,500 $= T_{999}$ is the last entry of block 999, so $m = 998$; position 499,501 opens block 1000 with $m = 0$; and the last line is the answer we computed by hand for $k = 10^9$ in Section 4.3.4.

4.3.8 An alternative: binary search on the block

The closed form needs a square root and the correctness argument about rounding. There is a second road that needs neither, and it uses only Chapter 2: **binary search on the answer**. The question "is $T_n \geq k$?" has the answer *no* for small n and *yes* for all n from some point on, and we want the first *yes*. That is a monotone yes/no question, which binary search answers in logarithmic time.

We need an upper end for the search. Since $T_{50000} = \frac{50000 \times 50001}{2} = 1{,}250{,}025{,}000 \geq 10^9$, the answer never exceeds 50,000, so we search in [1,50000]. The invariant is: *the answer lies in* $[lo, hi]$. If $T_{mid} \geq k$, then mid is large enough and the answer is at most mid, so set $hi = mid$; otherwise mid is too small and the answer is at least $mid + 1$, so set $lo = mid + 1$. When $lo = hi$ the window holds a single candidate, and by the invariant it is the answer.

To see it work by hand, start with the smaller window [1,8] and $k = 5$:

1. $lo = 1, hi = 8, mid = 1 + \lfloor 7/2 \rfloor = 4$. $T_4 = 10 \geq 5$, so $hi = 4$.
2. $lo = 1, hi = 4, mid = 1 + \lfloor 3/2 \rfloor = 2$. $T_2 = 3 < 5$, so $lo = 3$.
3. $lo = 3, hi = 4, mid = 3 + \lfloor 1/2 \rfloor = 3$. $T_3 = 6 \geq 5$, so $hi = 3$.
4. $lo = hi = 3$. The block is $n = 3$, and $m = 5 - T_2 - 1 = 5 - 3 - 1 = 1$. Answer `1 3` ✓

Listing 4.2 is the program. Note the midpoint `lo + (hi - lo) / 2`, which is the overflow-safe form of `(lo + hi) / 2` discussed in Section 2.14.6.

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.

```
#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;
}
```

On the edge-case input above it prints the same five lines, and on the positions 1 to 10 the same ten. The cost is about 16 iterations per query instead of $O(1)$ — irrelevant with ten queries, and a fair price for a program whose correctness argument is one invariant.

4.3.9 Checking against brute force

The proof in Section 4.3.5 is the real guarantee. But a proof can contain a slip, and a cheap experiment catches most slips. **Listing 4.3** builds the list the slow way — every value $2^m + 2^n$ with $n \leq 40$, sorted — and compares it with the formula for every one of its $820 = T_{40}$ positions. (Every value with $n \leq 40$ is smaller than every value with $n = 41$, by Claim 1, so the first 820 sorted values are exactly the ones generated.)

Listing 4.3 A brute-force cross-check for LDCSUMAB: the list built and sorted explicitly, compared with the formula at all 820 positions.

```
#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;
}
```

It prints

```
 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
```

The first ten rows reproduce **Table 4.2** exactly — values, m and n — and there are no mismatches anywhere. Building such a checker is the habit that turns “I think it is right” into “I checked it”.

4.3.10 Common mistakes on this problem

First: assuming the blocks do not interleave without proving it. They do not, and Claim 1 is the one-paragraph proof — but the proof must be written down, because “obviously the blocks do not overlap” is the kind of obvious thing that is sometimes false.

Second: using the floor where the ceiling belongs, or $>$ where $\geq$ belongs, when hunting for the block, so that k lands one block too early or too late. Test $k = 1,3,6,10$ and $k = 2,4,7,11$: every triangular number is a block boundary, and every triangular number plus one opens a new block.

Third: printing `n m` instead of `m n`. The problem says $m < n$ and the sample confirms the order.

Fourth: computing $n(n+1)$ in `int`. For k near 10^9 the product is about 2.0×10^9, a whisker below the `int` limit; any larger constraint overflows it silently. Use `long long`.

Fifth: building the list. It is the first idea everyone has, and Step 3 rejects it in ten seconds.

4.4 Problem 2 — TOHU: Help Tohu

The second judge problem, TOHU, is a SPOJ classical problem [4]. Where LDCSUMAB exercised the arithmetic series, TOHU exercises the other great summation tool of Chapter 3: the **telescoping sum** of Section 3.9.5. Its whole solution is five lines of algebra and a one-line program, but each of the five lines deserves care, and the problem hides a trap at $k = 1$ that catches careful people.

4.4.1 The problem in our own words

Tohu wants to study at a university and must first pass an entrance test in mathematics. He asks for help with one problem. A sequence of numbers $a_1, a_2, a_3, \ldots$ is not given to us directly. Instead we are told that, for every positive integer k, a certain weighted sum of its first k terms has a known value:

$$a_1 + 2a_2 + 3a_3 + \cdots + k\,a_k = \frac{k+1}{k+2} \quad \text{for every } k \geq 1 \tag{4.4}$$

We must compute the ordinary, unweighted sum of the first n terms, $S_n = a_1 + a_2 + \cdots + a_n$, for many values of n.

The input starts with the number of test cases T, at most 20,000; each following line holds one n. For each n we print S_n with exactly eleven digits after the decimal point. With the sample input `2`, `2`, `5` the expected output is `0.70833333333` and `0.73809523810`.

> **PROBLEM.** The sequence $a_1, a_2, \ldots$ satisfies Equation (4.4). For each of up to 20,000 queries n, print $S_n = a_1 + \cdots + a_n$ with eleven digits after the decimal point. The problem is usually posed with the note *time complexity* $O(1)$.

4.4.2 Steps 1 to 3: the formats, the symbols, the budget

Step 1 — the formats, copied.

- Input: line 1 holds T ($T \leq 20{,}000$). Then T lines, each holding one positive integer n.
- Output: for each n, one line with S_n printed with exactly 11 digits after the decimal point.
- Sample input `2 / 2 / 5`; sample output `0.70833333333 / 0.73809523810`.

Eleven digits is a lot. It rules out `float` at once, and it means the format string must be `%.11f`.

Step 2 — given and wanted.

- Given: $n \geq 1$; and, implicitly, the sequence through Equation (4.4).
- Wanted: $S_n = \sum_{k=1}^{n} a_k$.

There are really *two* unknowns here: first the terms a_k, which the problem hides behind a weighted sum, and then their plain sum. Naming both is what makes the plan visible: recover a_k, then add them up.

Step 3 — the budget. The statement sets no small bound on n, so we must be ready for large values. With up to 20,000 queries, adding up n terms per query costs $\Theta(Tn)$, which is 2×10^{13} if n can reach 10^9 — hopeless against **Table 4.1**. The note $O(1)$ says the same thing more bluntly: there is a closed form for S_n, and our job is to find it.

4.4.3 Step 4: the first cases by hand

Let us write W_k for the weighted sum on the left of Equation (4.4), so that $W_k = \frac{k+1}{k+2}$. Going from W_{k-1} to W_k adds exactly one new term, $k\,a_k$. So

$$k\,a_k = W_k - W_{k-1}, \qquad \text{with } W_0 = 0 \text{ (the empty sum).}$$

That single observation lets us compute the sequence one term at a time. Every subtraction below is done over a common denominator.

- $k = 1$: $W_1 = \frac{2}{3}$ and $W_0 = 0$, so $1 \cdot a_1 = \frac{2}{3}$ and $a_1 = \frac{2}{3}$. Then $S_1 = \frac{2}{3}$.
- $k = 2$: $W_2 = \frac{3}{4}$. $2a_2 = \frac{3}{4} - \frac{2}{3} = \frac{9}{12} - \frac{8}{12} = \frac{1}{12}$, so $a_2 = \frac{1}{24}$. $S_2 = \frac{2}{3} + \frac{1}{24} = \frac{16}{24} + \frac{1}{24} = \frac{17}{24}$.
- $k = 3$: $W_3 = \frac{4}{5}$. $3a_3 = \frac{4}{5} - \frac{3}{4} = \frac{16}{20} - \frac{15}{20} = \frac{1}{20}$, so $a_3 = \frac{1}{60}$. $S_3 = \frac{17}{24} + \frac{1}{60} = \frac{85}{120} + \frac{2}{120} = \frac{87}{120} = \frac{29}{40}$.
- $k = 4$: $W_4 = \frac{5}{6}$. $4a_4 = \frac{5}{6} - \frac{4}{5} = \frac{25}{30} - \frac{24}{30} = \frac{1}{30}$, so $a_4 = \frac{1}{120}$. $S_4 = \frac{29}{40} + \frac{1}{120} = \frac{87}{120} + \frac{1}{120} = \frac{88}{120} = \frac{11}{15}$.
- $k = 5$: $W_5 = \frac{6}{7}$. $5a_5 = \frac{6}{7} - \frac{5}{6} = \frac{36}{42} - \frac{35}{42} = \frac{1}{42}$, so $a_5 = \frac{1}{210}$. $S_5 = \frac{11}{15} + \frac{1}{210} = \frac{154}{210} + \frac{1}{210} = \frac{155}{210} = \frac{31}{42}$.

Table 4.3 collects the results.

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

Two things leap out of the table. First, the rows for S_2 and S_5 are exactly the sample outputs, so we have already solved the sample by hand. Second, from $k = 2$ onwards the terms are $\frac{1}{2 \cdot 3 \cdot 4}, \frac{1}{3 \cdot 4 \cdot 5}, \frac{1}{4 \cdot 5 \cdot 6}, \frac{1}{5 \cdot 6 \cdot 7}$: the conjecture is $a_k = \frac{1}{k(k+1)(k+2)}$. But $a_1 = \frac{2}{3}$, and $\frac{1}{1 \cdot 2 \cdot 3} = \frac{1}{6}$. The first term does *not* fit. A table that shows a pattern also shows where the pattern breaks, and that is precious information.

4.4.4 Steps 5 and 6: recovering a_k by differencing

Step 5 — the shape. This is *closed form from a sum*, twice over: first undo a sum to recover the terms, then sum the terms in closed form.

Step 6 — the formula for a_k, proved. Write Equation (4.4) at k and at $k-1$, one above the other, for any $k \geq 2$:

$$a_1 + 2a_2 + \cdots + (k-1)a_{k-1} + ka_k = \frac{k+1}{k+2}$$

$$a_1 + 2a_2 + \cdots + (k-1)a_{k-1} \qquad = \frac{k}{k+1}$$

(The second line is Equation (4.4) with $k-1$ in place of k: $\frac{(k-1)+1}{(k-1)+2} = \frac{k}{k+1}$.) Subtract the second line from the first. Every term on the left cancels except ka_k:

$$k\,a_k = \frac{k+1}{k+2} - \frac{k}{k+1}.$$

Put the right-hand side over the common denominator $(k+1)(k+2)$:

$$k\,a_k = \frac{(k+1)(k+1) - k(k+2)}{(k+1)(k+2)} = \frac{(k^2+2k+1) - (k^2+2k)}{(k+1)(k+2)} = \frac{1}{(k+1)(k+2)}.$$

Divide both sides by k:

$$a_k = \frac{1}{k(k+1)(k+2)} \quad \text{for } k \geq 2, \qquad a_1 = \frac{2}{3} \tag{4.5}$$

The formula for $k \geq 2$ is now a theorem, not a guess: the algebra holds for every $k \geq 2$ at once. And the algebra also *explains* why $k = 1$ is different. The subtraction used the condition at $k-1$, which exists only when $k-1 \geq 1$. If we applied the formula $W_{k-1} = \frac{k}{k+1}$ at $k = 1$ anyway, it would claim $W_0 = \frac{1}{2}$. But W_0 is the empty sum, which is 0. The formula overstates the previous weighted sum by $\frac{1}{2}$, so the pattern understates $1 \cdot a_1$ by $\frac{1}{2}$: the pattern gives $\frac{1}{1 \cdot 2 \cdot 3} = \frac{1}{6}$, and the truth is $\frac{1}{6} + \frac{1}{2} = \frac{1}{6} + \frac{3}{6} = \frac{4}{6} = \frac{2}{3}$. Nothing about a_1 is mysterious; it is simply the one term for which the differencing trick has no predecessor.

4.4.5 Summing by telescoping

We want

$$S_n = a_1 + \sum_{k=2}^{n} a_k = \frac{2}{3} + \sum_{k=2}^{n} \frac{1}{k(k+1)(k+2)}.$$

Chapter 3's telescoping sums need a term that is a *difference of consecutive values of one function*. The term $\frac{1}{k(k+1)(k+2)}$ does not look like that, but it becomes one. Compute the difference of $\frac{1}{k(k+1)}$ and its successor $\frac{1}{(k+1)(k+2)}$:

$$\frac{1}{k(k+1)} - \frac{1}{(k+1)(k+2)} = \frac{(k+2)-k}{k(k+1)(k+2)} = \frac{2}{k(k+1)(k+2)}.$$

Divide by 2:

$$\frac{1}{k(k+1)(k+2)} = \frac{1}{2}\left[\frac{1}{k(k+1)} - \frac{1}{(k+1)(k+2)}\right] \tag{4.6}$$

Now sum Equation (4.6) from $k = 2$ to $k = n$ and watch the middle terms cancel. Writing out the bracket for each k:

- $k = 2$: $\frac{1}{2\cdot 3} - \frac{1}{3\cdot 4}$
- $k = 3$: $\frac{1}{3\cdot 4} - \frac{1}{4\cdot 5}$
- $k = 4$: $\frac{1}{4\cdot 5} - \frac{1}{5\cdot 6}$
- $\vdots$
- $k = n$: $\frac{1}{n(n+1)} - \frac{1}{(n+1)(n+2)}$

Each negative term is cancelled by the positive term on the next line. Only the first positive term and the last negative term survive:

$$\sum_{k=2}^{n} \frac{1}{k(k+1)(k+2)} = \frac{1}{2}\left[\frac{1}{2\cdot 3} - \frac{1}{(n+1)(n+2)}\right] = \frac{1}{12} - \frac{1}{2(n+1)(n+2)}.$$

Therefore

$$\begin{aligned} S_n = \frac{2}{3} + \frac{1}{12} - \frac{1}{2(n+1)(n+2)} &= \frac{8}{12} + \frac{1}{12} - \frac{1}{2(n+1)(n+2)} \\ &= \frac{9}{12} - \frac{1}{2(n+1)(n+2)}, \end{aligned}$$

and since $\frac{9}{12} = \frac{3}{4}$,

$$S_n = \frac{3}{4} - \frac{1}{2(n+1)(n+2)} \tag{4.7}$$

Check against the table. $n = 2$: $\frac{3}{4} - \frac{1}{2\cdot 3\cdot 4} = \frac{3}{4} - \frac{1}{24} = \frac{18}{24} - \frac{1}{24} = \frac{17}{24}$ ✓, which is 0.708333 … and prints as `0.70833333333`. $n = 5$: $\frac{3}{4} - \frac{1}{2\cdot 6\cdot 7} = \frac{3}{4} - \frac{1}{84} = \frac{63}{84} - \frac{1}{84} = \frac{62}{84} = \frac{31}{42}$ ✓, which is 0.738095238095 … and rounds to `0.73809523810`.

A second road: partial fractions. Another standard way to split $\frac{1}{k(k+1)(k+2)}$ is to look for constants A, B, C with

$$\frac{1}{k(k+1)(k+2)} = \frac{A}{k} + \frac{B}{k+1} + \frac{C}{k+2}.$$

Multiply through by $k(k+1)(k+2)$:

$$\begin{aligned} 1 &= A(k+1)(k+2) + Bk(k+2) + Ck(k+1) \\ &= (A+B+C)k^2 + (3A+2B+C)k + 2A. \end{aligned}$$

Two polynomials are equal for every k only if their coefficients match, so

$$A + B + C = 0, \qquad 3A + 2B + C = 0, \qquad 2A = 1.$$

From the third, $A = 1/2$. Subtracting the first equation from the second gives $2A + B = 0$, so $B = -1$. Then $C = -A - B = -1/2 + 1 = 1/2$. So

$$\frac{1}{k(k+1)(k+2)} = \frac{1}{2k} - \frac{1}{k+1} + \frac{1}{2(k+2)}.$$

Summed from 2 to n, these three pieces also telescope, though the cancellation is spread over three columns instead of two and takes more bookkeeping to see. Grouping them as $\frac{1}{2}\left[\frac{1}{k} - \frac{1}{k+1}\right] - \frac{1}{2}\left[\frac{1}{k+1} - \frac{1}{k+2}\right]$ turns the partial-fraction form back into Equation (4.6). Both roads arrive at Equation (4.7); the first is shorter, the second is the one that works for any rational function, and knowing both is worth the extra page.

4.4.6 Proof of correctness

The derivation above is already a proof, since every step is an identity. But a result reached by clever regrouping is worth confirming by a second, completely mechanical method, and Chapter 3's induction is exactly that.

THEOREM. For every $n \geq 1$, $S_n = \frac{3}{4} - \frac{1}{2(n+1)(n+2)}$.

Proof by induction on n.

Base case, $n = 1$. $S_1 = a_1 = \frac{2}{3}$. The formula gives $\frac{3}{4} - \frac{1}{2\cdot 2\cdot 3} = \frac{3}{4} - \frac{1}{12} = \frac{9}{12} - \frac{1}{12} = \frac{8}{12} = \frac{2}{3}$. ✓ Notice that the formula covers $n = 1$ even though a_1 is the exceptional term; the exception has been absorbed into the constant $\frac{3}{4}$.

Inductive step. Let $n \geq 2$ and assume $S_{n-1} = \frac{3}{4} - \frac{1}{2n(n+1)}$. Since $n \geq 2$, Equation (4.5) gives $a_n = \frac{1}{n(n+1)(n+2)}$. Then

$$S_n = S_{n-1} + a_n = \frac{3}{4} - \frac{1}{2n(n+1)} + \frac{1}{n(n+1)(n+2)}.$$

Put the last two terms over the common denominator $2n(n+1)(n+2)$:

$$\begin{aligned} -\frac{n+2}{2n(n+1)(n+2)} + \frac{2}{2n(n+1)(n+2)} &= -\frac{(n+2)-2}{2n(n+1)(n+2)} \\ &= -\frac{n}{2n(n+1)(n+2)} = -\frac{1}{2(n+1)(n+2)}. \end{aligned}$$

So $S_n = \frac{3}{4} - \frac{1}{2(n+1)(n+2)}$, which is the claim for n. ■

The theorem also tells us something the problem never asked: as n grows, $\frac{1}{2(n+1)(n+2)}$ shrinks to 0 and S_n climbs towards $\frac{3}{4}$ without ever reaching it. Every answer the program prints lies in $\left[\frac{2}{3}, \frac{3}{4}\right)$.

4.4.7 Complexity, precision and overflow

Time and space. Equation (4.7) costs two additions, two multiplications, one division and one subtraction: $\Theta(1)$ per query, $\Theta(T)$ in total, $O(1)$ space. With $T = 20{,}000$ that is about 10^5 operations — a thousandth of the budget in **Table 4.1**.

Precision. The answer lies below 0.75 and must be printed to eleven decimal places, so we need about eleven correct significant digits. A `double` carries fifteen to sixteen, so the rounding error of the handful of operations above is far below the last printed digit. A `float`, with about seven, would fail on the sample itself.

Overflow. The denominator $2(n+1)(n+2)$ grows like $2n^2$. If `n` were an `int` and we wrote `2 * (n+1) * (n+2)` in integer arithmetic, the product would pass the `int` limit of 2,147,483,647 as soon as $n = 32{,}767$, where it would be $2 \times 32{,}768 \times 32{,}769 = 2{,}147{,}549{,}184$ — silently producing garbage. The program below avoids the question entirely by converting each factor to `double` *before* multiplying. A `double` can hold $2(n+1)(n+2)$ even for $n = 10^9$, where it is about 2×10^{18}; and at that size $\frac{1}{2(n+1)(n+2)} \approx 5 \times 10^{-19}$ is far below the eleventh decimal place, so the program correctly prints `0.75000000000`.

4.4.8 Step 7: the program

Pseudocode 4.2 is the whole algorithm.

Pseudocode 4.2 The TOHU algorithm: one evaluation of the closed form Equation (4.7) per query.

```
TOHU()
read T
for q = 1 to T
    read n
    print 3/4 - 1 / (2(n+1)(n+2)) with 11 decimal places
```

Figure 4.4 shows another correct C++ solution. It treats $n = 1$ as a special case and prints $\frac{2}{3}$ directly.

```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 4.4 An alternative TOHU program with a separate branch `if (n==1) printf("%.11lf\n", 2.0/3.0);` and a general branch that evaluates $\frac{1}{12} - \frac{1}{2(n+1)(n+2)} + \frac{2}{3}$.

That branch does no harm, but it is unnecessary: the base case of the induction in Section 4.4.6 showed that Equation (4.7) already gives $\frac{2}{3}$ at $n = 1$. Noticing a redundant special case, and saying why it is redundant, is exactly the kind of checking Step 6 asks for. Notice also, in **Figure 4.4**, the cast in the denominator `(double) 2 * (n+1) * (n+2)`: converting the leading `2` to `double` forces the first multiplication into floating point, so `(n+1)` and `(n+2)` are promoted before they are multiplied and nothing is ever multiplied as two `int`s. It is a correct but fragile way to avoid the overflow of Section 4.4.7. **Listing 4.4** drops the branch and casts every factor explicitly.

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;
}
```

Compile it exactly as in Section 4.3.7 — `g++ -std=c++17 -O2 -Wall -o tohu tohu.cpp` — and run it on the sample input `2`, `2`, `5`. It prints

```
0.70833333333
0.73809523810
```

which is the sample output. On the six queries $n = 1,2,3,4,10,10^9$ it prints

```
0.66666666667
0.70833333333
0.72500000000
0.73333333333
0.74621212121
0.75000000000
```

The first four lines are the decimal column of **Table 4.3**; the fifth is $\frac{3}{4} - \frac{1}{2 \cdot 11 \cdot 12} = \frac{3}{4} - \frac{1}{264} = \frac{197}{264}$; the sixth shows the limit $\frac{3}{4}$ reached to eleven places.

4.4.9 An alternative: exact fractions, with no formula assumed

The closed form is the solution. But suppose you did not trust the algebra, or had not found it yet. You could still compute the sequence *directly from the definition*, using the rule $k\,a_k = W_k - W_{k-1}$ of Section 4.4.3, in exact rational arithmetic so that no rounding can hide a mistake. **Listing 4.5** does exactly that. It stores each fraction as a numerator and denominator reduced by their greatest common divisor (`std::gcd`, available since C++17), recovers $a_1, \ldots, a_{10}$ from Equation (4.4) alone, accumulates S_k, and prints the exact fraction next to the decimal value and the closed form.

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.

```
#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;
}
```

It prints

```
 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
```

Every exact fraction for $k \leq 5$ matches **Table 4.3**, the a_k column confirms Equation (4.5) out to $k = 10$ — $a_{10} = \frac{1}{10 \cdot 11 \cdot 12} = \frac{1}{1320}$ — and the last two columns agree to all eleven places. As a *solution* this approach is useless for large n: it is $\Theta(n)$ per query, and the denominators grow like n^3 until they overflow `long long`. As a *check* it is excellent, because it assumes nothing that we derived.

4.4.10 Common mistakes on this problem

First: computing S_n by looping k from 1 to n. With twenty thousand queries and large n that is hopeless, and the problem said so by writing $O(1)$.

Second: getting the differencing backward — subtracting the k equation from the $k-1$ equation and ending up with $-ka_k$. Write the two equations one above the other, as in Section 4.4.4, and the cancellation is visible.

Third: assuming $a_k = \frac{1}{k(k+1)(k+2)}$ for $k = 1$ too. That gives $S_n = \frac{1}{4} - \frac{1}{2(n+1)(n+2)}$, which is wrong by exactly $\frac{1}{2}$. The hand table of Step 4 catches this in one line.

Fourth: printing with a `float`, or with `%.10f`, or with `%lf` on an old compiler that mishandles it. Use `double` and `%.11f`.

Fifth: multiplying `(n+1)*(n+2)` in `int` arithmetic, which overflows from $n = 32{,}767$ on.

Sixth: writing `3/4` in C++ and getting `0`. Integer division truncates; write `0.75` or `3.0/4.0`.

4.5 Worked Problems on Correctness: Invariants and Traces

The two judge problems showed the whole method at full length. The twelve worked problems that follow are shorter, and each one aims at a single tool from Chapters 1 to 3. They all have the same shape, so that you always know where you are:

- **Problem** — the statement.
- **What it tests** — the tool, and where in the book it was introduced.
- **Solution** — step by step, with every arithmetic step shown.
- **Answer** — the result in one line.
- **Program** — a complete C++17 program where running one teaches something, with the output it really printed.
- **Check yourself** — a variant for you to do alone, with its answer.

This section works three problems on *correctness*: proving a loop right, and following one exactly.

4.5.1 Worked Problem A — a loop invariant for Horner's rule

Problem. A polynomial of degree n is $P(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$. Store its coefficients in an array $A[0..n]$, with $A[k] = a_k$. The procedure in **Pseudocode 4.3**, known as **Horner's rule**, claims to evaluate $P(x)$ with only $n + 1$ multiplications. State a loop invariant for it, prove the invariant, and use it to prove that the procedure returns $P(x)$. Then trace it on $P(x) = 2x^3 - 6x^2 + 2x - 1$ at $x = 3$.

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
```

What it tests. The three-part loop-invariant proof of Section 2.9 (initialization, maintenance, termination), applied to a loop you have not seen proved, and the index manipulation of Chapter 3's summations.

Solution. Where does the idea come from? Factor x out of everything except the constant term, again and again:

$$a_0 + a_1x + a_2x^2 + a_3x^3 = a_0 + x(a_1 + x(a_2 + x\,a_3)).$$

Evaluating from the inside out needs one multiplication per level. The loop does exactly that: at each step it takes what it has, multiplies by x, and adds the next coefficient down. So at any moment p is the polynomial built from the coefficients processed *so far*, the highest ones, with the lowest of them as its constant term. That sentence is the invariant; now write it in symbols.

> **LOOP INVARIANT (Horner's rule).** At the start of each iteration of the `for` loop of lines 2–3, with loop index i,

$$p = \sum_{k=0}^{n-(i+1)} A\,[k + i + 1]\,x^k \tag{4.8}$$

In words: p is the polynomial whose coefficients are $A[i + 1], A[i + 2], \ldots, A[n]$, with $A[i + 1]$ as the constant term. Now the three obligations.

Initialization. Before the first iteration, $i = n$. The sum in Equation (4.8) runs from $k = 0$ to $k = n - (n + 1) = -1$. A sum whose upper limit is below its lower limit is empty, and an empty sum equals 0. Line 1 set $p = 0$. ✓

Maintenance. Assume Equation (4.8) holds at the start of the iteration with index i. Line 3 computes the new value

$$p' = A[i] + x \cdot \sum_{k=0}^{n-(i+1)} A\,[k + i + 1]\,x^k = A[i] + \sum_{k=0}^{n-i-1} A\,[k + i + 1]\,x^{k+1}.$$

Rename the summation index: put $j = k + 1$, so j runs from 1 to $n - i$ and $A[k + i + 1] = A[j + i]$:

$$p' = A[i]\,x^0 + \sum_{j=1}^{n-i} A\,[j+i]\,x^j = \sum_{j=0}^{n-i} A\,[j+i]\,x^j,$$

because the lone term $A[i]$ is exactly the $j = 0$ term of the same sum. Now the loop decrements the index: the next iteration has index $i' = i - 1$. Substituting $i = i' + 1$ gives $p' = \sum_{j=0}^{n-(i'+1)} A\,[j + i' + 1]\,x^j$, which is Equation (4.8) for the index i'. ✓

Termination. The loop runs for $i = n, n-1, \ldots, 0$ and stops when i reaches -1, after exactly $n + 1$ iterations. Substituting $i = -1$ into Equation (4.8):

$$p = \sum_{k=0}^{n-0} A\,[k+0]\,x^k = \sum_{k=0}^{n} a_k\,x^k = P(x).$$

Line 4 returns p, which is $P(x)$. ✓ ■

The trace. For $P(x) = 2x^3 - 6x^2 + 2x - 1$ the coefficients are $A[0] = -1$, $A[1] = 2$, $A[2] = -6$, $A[3] = 2$, and $n = 3$, $x = 3$. **Table 4.4** follows the loop, and its last column evaluates the right-hand side of Equation (4.8) independently, so that the invariant can be seen holding at every step.

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	$-1 + 3 \cdot 2 = 5$	$A[0] + 3A[1] + 9A[2] + 27A[3] = -1 + 6 - 54 + 54 = 5$

Cost. Line 3 runs $n + 1$ times, with one multiplication and one addition each: $\Theta(n)$. The naive method computes each power x^k from scratch with k multiplications and then multiplies by a_k, costing $\sum_{k=0}^{n}(k+1) = \frac{(n+1)(n+2)}{2} = \Theta(n^2)$ multiplications — the arithmetic series again.

Answer. The invariant is Equation (4.8); at termination it gives $p = P(x)$. For the example, $P(3) = 5$, computed with 4 multiplications.

Program. **Listing 4.6** runs Horner's rule with a trace, runs the naive method, and counts multiplications for both.

Listing 4.6 Horner's rule against naive evaluation, with a trace of the loop and a count of multiplications.

```
#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
```

The trace reproduces **Table 4.4**, and the counts are $n + 1$ against $\frac{(n+1)(n+2)}{2}$: for degree 1000, 1,001 multiplications against 501,501.

> **CHECK YOURSELF.** The loop `r = 1; for i = 1 to n: r = r * x` computes x^n. State its invariant and prove it.
>
> *Answer.* At the start of the iteration with index i, $r = x^{i-1}$. Initialization: $i = 1$, $r = 1 = x^0$. Maintenance: $r \cdot x = x^{i-1} \cdot x = x^i$, which is the invariant for the next index $i + 1$. Termination: the loop stops at $i = n + 1$, where the invariant gives $r = x^n$.

4.5.2 Worked Problem B — tracing Insertion Sort

Problem. Trace INSERTION-SORT (**Pseudocode 4.4**) on $A = \langle 8,3,6,3,9,1 \rangle$. After each iteration of the outer loop give the array, the number of key comparisons $A[i] > key$ evaluated, and the number of elements shifted. Compare the total number of shifts with the number of *inversions* of A — pairs of positions $p < q$ with $A[p] > A[q]$.

Pseudocode 4.4 Insertion Sort, as in Section 2.8.2, with indices running from 1.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      i = j - 1
4      while i > 0 and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

What it tests. Following an algorithm exactly (Section 2.8.3), the meaning of the cost t_j in the analysis of Section 2.10, and the stability of the strict `>` on line 4.

Solution. Work one outer iteration at a time. The part of the array to the left of position j is always sorted — that is the invariant of Section 2.9.

- $j = 2$, $key = 3$. Compare $A[1] = 8 > 3$: yes, shift 8 to position 2, $i = 0$. The test $i > 0$ fails, so no further comparison. Place 3 at position 1. Array $\langle 3,8,6,3,9,1 \rangle$: 1 comparison, 1 shift.
- $j = 3$, $key = 6$. $8 > 6$: shift. $3 > 6$: no, stop. Place 6 at position 2. Array $\langle 3,6,8,3,9,1 \rangle$: 2 comparisons, 1 shift.
- $j = 4$, $key = 3$. $8 > 3$: shift. $6 > 3$: shift. $3 > 3$: **no** — equal is not greater — stop. Place the key at position 2. Array $\langle 3,3,6,8,9,1 \rangle$: 3 comparisons, 2 shifts. The key stopped *behind* the equal 3, so the two 3s keep their original order; this is the stability that the strict `>` buys.
- $j = 5$, $key = 9$. $8 > 9$: no, stop immediately. Array unchanged: 1 comparison, 0 shifts. An element already in place costs one comparison — the best case of Section 2.11.1 in miniature.

- $j = 6$, $key = 1$. 9,8,6,3,3 are all greater than 1: five shifts, and then $i = 0$ ends the loop. Array ⟨1,3,3,6,8,9⟩: 5 comparisons, 5 shifts. This is the worst case for one iteration: the key travels the whole sorted prefix.

Totals: $1 + 2 + 3 + 1 + 5 = 12$ comparisons and $1 + 1 + 2 + 0 + 5 = 9$ shifts.

Inversions. Count pairs $p < q$ with $A[p] > A[q]$ in the *original* array ⟨8,3,6,3,9,1⟩, going left to right:

- 8 is greater than the later 3,6,3,1: 4 inversions.
- The first 3 is greater than the later 1: 1 (not greater than the second 3).
- 6 is greater than the later 3,1: 2.
- The second 3 is greater than the later 1: 1.
- 9 is greater than the later 1: 1.

Total $4 + 1 + 2 + 1 + 1 = 9$ — exactly the number of shifts. That is no coincidence: each shift on line 5 moves a larger element past a smaller one that lay to its right, which removes exactly one inversion and creates none. The array is sorted when no inversions remain, so **the number of shifts equals the number of inversions**, always. **Table 4.5** summarizes.

Table 4.5 Insertion Sort on ⟨8,3,6,3,9,1⟩. 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	⟨3,8,6,3,9,1⟩	1	1	2
3	6	⟨3,6,8,3,9,1⟩	2	1	2
4	3	⟨3,3,6,8,9,1⟩	3	2	3
5	9	⟨3,3,6,8,9,1⟩	1	0	1
6	1	⟨1,3,3,6,8,9⟩	5	5	6
		total	**12**	**9**	**14**

The last column connects the trace to the analysis of Section 2.10, where t_j is the number of times the `while` test on line 4 is *executed* in iteration j. Every shift is preceded by one successful test, and the loop ends with exactly one failing test — either $A[i] > key$ is false, or $i > 0$ is false. So t_j is always the number of shifts plus one: 2,2,3,1,6, summing to 14.

Answer. Final array ⟨1,3,3,6,8,9⟩; 12 comparisons, 9 shifts, 9 inversions, $\sum_{j=2}^{6} t_j = 14$.

Program. **Listing 4.7** performs the same trace and counts the inversions independently.

Listing 4.7 Insertion Sort with a per-iteration trace, and an independent inversion count for comparison.

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

void show(const vector<int>& A) {                  // print A[1..n]
    printf("[");
    for (size_t t = 1; t < A.size(); ++t) printf("%s%d", t > 1 ? ", " : "", A[t]);
    printf("]");
}

int main() {
    vector<int> A = {0, 8, 3, 6, 3, 9, 1};         // A[0] unused: indices run 1..n as in
the pseudocode
    int n = (int)A.size() - 1;
    long long totalCmp = 0, totalShift = 0;
    printf("start       "); show(A); printf("\n");
    for (int j = 2; j <= n; ++j) {
        int key = A[j], i = j - 1, cmp = 0, shift = 0;
        while (i > 0) {
            ++cmp;                                  // one test of A[i] > key
            if (A[i] <= key) break;
            A[i + 1] = A[i]; --i; ++shift;
        }
        A[i + 1] = key;
        totalCmp += cmp; totalShift += shift;
        printf("j=%d key=%d  ", j, key); show(A);
        printf("  comparisons=%d shifts=%d\n", cmp, shift);
    }
    // Count inversions of the ORIGINAL array directly, for comparison.
    vector<int> B = {0, 8, 3, 6, 3, 9, 1};
    int inv = 0;
    for (int p = 1; p <= n; ++p)
        for (int q = p + 1; q <= n; ++q)
            if (B[p] > B[q]) ++inv;
    printf("total comparisons = %lld, total shifts = %lld, inversions = %d\n",
           totalCmp, totalShift, inv);
    return 0;
}
```

```
start       [8, 3, 6, 3, 9, 1]
j=2 key=3  [3, 8, 6, 3, 9, 1]  comparisons=1 shifts=1
j=3 key=6  [3, 6, 8, 3, 9, 1]  comparisons=2 shifts=1
j=4 key=3  [3, 3, 6, 8, 9, 1]  comparisons=3 shifts=2
j=5 key=9  [3, 3, 6, 8, 9, 1]  comparisons=1 shifts=0
j=6 key=1  [1, 3, 3, 6, 8, 9]  comparisons=5 shifts=5
total comparisons = 12, total shifts = 9, inversions = 9
```

CHECK YOURSELF. Trace Insertion Sort on $\langle 4,1,3,9,7\rangle$. How many shifts, and how many inversions?

Answer. The arrays after $j = 2,3,4,5$ are $\langle 1,4,3,9,7\rangle$, $\langle 1,3,4,9,7\rangle$, $\langle 1,3,4,9,7\rangle$, $\langle 1,3,4,7,9\rangle$. Shifts $1 + 1 + 0 + 1 = 3$; comparisons $1 + 2 + 1 + 2 = 6$. The inversions are (4,1), (4,3) and (9,7): three, equal to the shifts.

4.5.3 Worked Problem C — tracing binary search

Problem. The sorted array $A[1..15] = \langle 3,8,11,15,19,22,27,31,36,40,44,51,57,62,70\rangle$ is searched with BINARY-SEARCH (**Pseudocode 4.5**). Trace the search for $q = 44$ and for $q = 20$, giving $left$, $right$ and mid at every probe. How many probes can a search in this array ever need?

Pseudocode 4.5 Binary search, as in Section 2.14.3, returning an index of q or NIL.

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

What it tests. The binary-search procedure and its window invariant (Section 2.14.3), and the $\lfloor \lg n \rfloor + 1$ bound on the number of probes (Section 2.14.5).

Solution. The window $[left, right]$ is where q could still be. At each probe compute mid, compare, and discard half.

Search for $q = 44$.

1. $left = 1, right = 15$. $mid = \lfloor 16/2 \rfloor = 8$. $A[8] = 31 < 44$, so q is to the right: $left = 9$.
2. $left = 9, right = 15$. $mid = \lfloor 24/2 \rfloor = 12$. $A[12] = 51 > 44$: $right = 11$.
3. $left = 9, right = 11$. $mid = \lfloor 20/2 \rfloor = 10$. $A[10] = 40 < 44$: $left = 11$.
4. $left = 11, right = 11$. $mid = 11$. $A[11] = 44$. Found; return 11.

Search for $q = 20$.

1. $left = 1, right = 15, mid = 8$. $A[8] = 31 > 20$: $right = 7$.
2. $left = 1, right = 7, mid = \lfloor 8/2 \rfloor = 4$. $A[4] = 15 < 20$: $left = 5$.
3. $left = 5, right = 7, mid = 6$. $A[6] = 22 > 20$: $right = 5$.
4. $left = 5, right = 5, mid = 5$. $A[5] = 19 < 20$: $left = 6$.
5. Now $left = 6 > right = 5$: the window is empty, so return NIL.

The invariant explains the NIL. At the end, the window is empty, and 20 would have had to sit between $A[5] = 19$ and $A[6] = 22$ — where there is no position at all. **Table 4.6** lays both searches side by side.

Table 4.6 Binary search in a 15-element array for a present key (44) and an absent key (20). The window shrinks 15 → 7 → 3 → 1 → 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

The worst case. Look at the last column. With $15 = 2^4 - 1$ elements, each probe that fails removes mid and leaves one of two halves of exactly 7; then 3; then 1; then 0. In general a window of size w leaves at most $\lfloor w/2 \rfloor$, so after p probes at most $\lfloor n/2^p \rfloor$ remain, and the search ends after at most $\lfloor \lg n \rfloor + 1$ probes. For $n = 15$: $\lfloor \lg 15 \rfloor + 1 = 3 + 1 = 4$, since $2^3 = 8 \leq 15 < 16 = 2^4$.

Answer. $q = 44$ is found at index 11 after 4 probes; $q = 20$ is reported absent after 4 probes; no search in this array needs more than 4.

Program. **Listing 4.8** prints the same trace.

Listing 4.8 Binary search with a line of output per probe, on the array of Worked Problem C.

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

// BINARY-SEARCH(A, q) on A[1..n]; prints one line per probe. Returns 0 for NIL.
int binarySearch(const vector<int>& A, int q) {
    int left = 1, right = (int)A.size() - 1, probe = 0;
    while (left <= right) {
        int mid = left + (right - left) / 2;      // same value as (left+right)/2, without
overflow
        ++probe;
        printf("  probe %d: left=%2d right=%2d mid=%2d A[mid]=%2d  ", probe, left, right,
mid, A[mid]);
        if (A[mid] == q) { printf("equal -> found at %d\n", mid); return mid; }
        if (A[mid] > q) { right = mid - 1; printf("too big   -> right=%d\n", right); }
        else            { left = mid + 1;  printf("too small -> left=%d\n", left); }
    }
    printf("  window empty (left=%d > right=%d) -> NIL\n", left, right);
    return 0;
}

int main() {
    vector<int> A = {0, 3, 8, 11, 15, 19, 22, 27, 31, 36, 40, 44, 51, 57, 62, 70};  // A[0]
unused
    for (int q : {44, 20}) {
        printf("search for %d\n", q);
        int r = binarySearch(A, q);
        printf("  result: %d\n", r);
    }
    return 0;
}
```

```
search for 44
  probe 1: left= 1 right=15 mid= 8 A[mid]=31  too small -> left=9
  probe 2: left= 9 right=15 mid=12 A[mid]=51  too big   -> right=11
  probe 3: left= 9 right=11 mid=10 A[mid]=40  too small -> left=11
  probe 4: left=11 right=11 mid=11 A[mid]=44  equal -> found at 11
  result: 11
search for 20
  probe 1: left= 1 right=15 mid= 8 A[mid]=31  too big   -> right=7
  probe 2: left= 1 right= 7 mid= 4 A[mid]=15  too small -> left=5
  probe 3: left= 5 right= 7 mid= 6 A[mid]=22  too big   -> right=5
  probe 4: left= 5 right= 5 mid= 5 A[mid]=19  too small -> left=6
  window empty (left=6 > right=5) -> NIL
  result: 0
```

CHECK YOURSELF. In the same array, trace the searches for $q = 70$ and $q = 3$.

Answer. For 70: $mid = 8$ (31, go right), $mid = 12$ (51, go right), $mid = 14$ (62, go right), $mid = 15$ (70, found) — 4 probes. For 3: $mid = 8$ (31, go left), $mid = 4$ (15, go left), $mid = 2$ (8, go left), $mid = 1$ (3, found) — 4 probes. The two ends of the array are as expensive as anything else, never more.

4.6 Worked Problems on Counting and Growth

The second group of worked problems practices the arithmetic of efficiency: counting exactly how many steps a loop takes, proving asymptotic statements with explicit constants, and placing functions in order of growth.

4.6.1 Worked Problem D — the exact step count of a triple loop

Problem. How many times does line 5 of LOOPS (**Pseudocode 4.6**) execute, as an exact function of n? Give the closed form and its Θ class.

Pseudocode 4.6 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
```

What it tests. Turning nested loops into nested sums (Section 2.10), and evaluating the sums with the formulas of Section 3.9.

Solution. Work from the inside out. Every loop becomes a sum, and the body of the innermost loop contributes 1 per execution.

The k loop. For fixed i and j, k runs from i to j: that is $j - i + 1$ values. (A loop from a to b runs $b - a + 1$ times when $b \geq a$. Check with $a = b$: once.)

The j loop. For fixed i, j runs from i to n, and the k loop contributes $j - i + 1$ each time. As j goes $i, i + 1, \ldots, n$, the quantity $t = j - i + 1$ goes $1,2, \ldots, n - i + 1$. So the j loop contributes

$$\sum_{j=i}^{n}(j - i + 1) = \sum_{t=1}^{n-i+1} t = \frac{(n - i + 1)(n - i + 2)}{2},$$

by the arithmetic series with $n - i + 1$ in place of n.

The i loop. Sum over i from 1 to n. Substitute $m = n - i + 1$: as i runs $1,2, \ldots, n$, m runs $n, n - 1, \ldots, 1$ — the same values in reverse order, and order does not matter in a sum. So the total is

$$\sum_{i=1}^{n} \frac{(n - i + 1)(n - i + 2)}{2} = \sum_{m=1}^{n} \frac{m(m + 1)}{2} = \frac{1}{2}\left(\sum_{m=1}^{n} m^2 + \sum_{m=1}^{n} m\right).$$

The two standard sums. $\sum_{m=1}^{n} m = \frac{n(n+1)}{2}$ is Equation (4.1). For the squares, the reference card of Section 3.9 gives $\sum_{m=1}^{n} m^2 = \frac{n(n+1)(2n+1)}{6}$; since this chapter consolidates, let us derive it too, by telescoping. Expand $(m + 1)^3 - m^3 = 3m^2 + 3m + 1$ and sum both sides for $m = 1$ to n. The left side telescopes to $(n + 1)^3 - 1$. So

$$(n + 1)^3 - 1 = 3\sum_{m=1}^{n} m^2 + 3 \cdot \frac{n(n + 1)}{2} + n.$$

Move the known terms across, using $(n+1)^3 - 1 - n = (n+1)^3 - (n+1) = (n+1)((n+1)^2 - 1) = (n+1)(n^2+2n)$:

$$3\sum_{m=1}^{n} m^2 = (n+1)(n^2+2n) - \frac{3n(n+1)}{2} = (n+1)\left(n^2+2n-\frac{3n}{2}\right)$$
$$= (n+1)\left(n^2+\frac{n}{2}\right) = \frac{n(n+1)(2n+1)}{2},$$

and dividing by 3 gives $\sum m^2 = \frac{n(n+1)(2n+1)}{6}$.

Putting it together.

$$\frac{1}{2}\left(\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}\right) = \frac{1}{2}\cdot\frac{n(n+1)}{6}((2n+1)+3)$$
$$= \frac{1}{2}\cdot\frac{n(n+1)(2n+4)}{6} = \frac{n(n+1)(n+2)}{6}.$$

$$\text{executions of line 5} = \sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1 = \frac{n(n+1)(n+2)}{6} \tag{4.9}$$

Check by hand at $n = 4$. Table 4.7 lists every (i, j) pair and the number of k values it produces.

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 $\frac{4\cdot5\cdot6}{6} = 20$.

i	Contributions *j−i+1* for *j=i,...,4*	Subtotal
1	$1+2+3+4$	10
2	$1+2+3$	6
3	$1+2$	3
4	1	1
	total	**20**

The formula gives $\frac{4\cdot5\cdot6}{6} = \frac{120}{6} = 20$ ✓. The table also shows the structure of the answer at a glance: each row is a triangular number, and adding up triangular numbers gives the so-called *tetrahedral* numbers 1,4,10,20,35, … — the number of oranges in a triangular pyramid.

Asymptotics. $\frac{n(n+1)(n+2)}{6} = \frac{n^3+3n^2+2n}{6}$. The leading term is $\frac{n^3}{6}$, so the count is $\Theta(n^3)$, with leading constant $\frac{1}{6}$. Compared with a plain triple loop in which every index runs from 1 to n, which executes n^3 times, LOOPS does about one sixth of the work — the same order of growth, a very different constant.

Answer. Line 5 executes exactly $\frac{n(n+1)(n+2)}{6}$ times, which is $\Theta(n^3)$.

Program. Listing 4.9 runs the loop and compares with Equation (4.9).

Listing 4.9 Counting the executions of the innermost line of LOOPS and comparing with $\frac{n(n+1)(n+2)}{6}$.

```
#include <cstdio>
#include <initializer_list>
using namespace std;
typedef long long LL;

// Executes line 5 of LOOPS(n) and counts how many times it runs.
LL loops(LL n) {
    LL count = 0;
    for (LL i = 1; i <= n; ++i)
        for (LL j = i; j <= n; ++j)
            for (LL k = i; k <= j; ++k)
                count = count + 1;
    return count;
}

int main() {
    printf("    n        count   n(n+1)(n+2)/6\n");
    for (LL n : {1LL, 2LL, 3LL, 4LL, 5LL, 6LL, 10LL, 100LL, 500LL}) {
        LL formula = n * (n + 1) * (n + 2) / 6;
        printf("%5lld  %11lld  %14lld  %s\n", n, loops(n), formula,
               loops(n) == formula ? "equal" : "DIFFERENT");
    }
    return 0;
}
```

```
    n        count   n(n+1)(n+2)/6
    1            1              1  equal
    2            4              4  equal
    3           10             10  equal
    4           20             20  equal
    5           35             35  equal
    6           56             56  equal
   10          220            220  equal
  100       171700         171700  equal
  500     20958500       20958500  equal
```

CHECK YOURSELF. How many times does the body of `for i = 1 to n: for j = i + 1 to n: body` execute? And the body of `i = n; while i > 1: i = floor(i / 2)`?

Answer. The first: for each i the inner loop runs $n - i$ times, so the total is $\sum_{i=1}^{n}(n-i) = (n-1) + (n-2) + \cdots + 0 = \frac{n(n-1)}{2}$, the number of pairs. The second: each iteration halves i (rounding down), and the loop stops when i reaches 1, which takes exactly $\lfloor \lg n \rfloor$ iterations — for $n = 10$, $10 \to 5 \to 2 \to 1$, three iterations, and $\lfloor \lg 10 \rfloor = 3$.

4.6.2 Worked Problem E — asymptotic claims with explicit constants

Problem. Prove or disprove each claim, giving explicit constants where the claim is true and a contradiction where it is false.

(a) $5n^2 - 3n + 12 = \Theta(n^2)$. (b) $2^{n+1} = O(2^n)$. (c) $2^{2n} = O(2^n)$. (d) $\lg(n!) = \Theta(n \lg n)$.

What it tests. The definitions of Chapter 3, used literally. Recall that $f(n) = \Theta(g(n))$ means there exist positive constants c_1, c_2 and n_0 with

$$0 \le c_1\, g(n) \le f(n) \le c_2\, g(n) \quad \text{for all } n \ge n_0 \tag{4.10}$$

and $f(n) = O(g(n))$ is the right-hand half of Equation (4.10) alone. A proof must *name* the constants; a disproof must show that no constants can work.

Solution (a). *Upper bound.* We want $5n^2 - 3n + 12 \le c_2 n^2$. Try $c_2 = 5$: the inequality becomes $-3n + 12 \le 0$, that is $12 \le 3n$, that is $n \ge 4$. So $c_2 = 5$ works for all $n \ge 4$.

Lower bound. We want $c_1 n^2 \leq 5n^2 - 3n + 12$. Try $c_1 = 4$: the inequality becomes $0 \leq n^2 - 3n + 12$. Complete the square: $n^2 - 3n + 12 = \left(n - \frac{3}{2}\right)^2 - \frac{9}{4} + 12 = \left(n - \frac{3}{2}\right)^2 + \frac{39}{4}$, which is positive for every n. So $c_1 = 4$ works for all $n \geq 1$.

Together: $c_1 = 4, c_2 = 5, n_0 = 4$ satisfy Equation (4.10). The threshold n_0 does real work: at $n = 3$ the upper inequality fails, since $f(3) = 45 - 9 + 12 = 48 > 45 = 5 \cdot 3^2$. Claim (a) is **true**.

Solution (b). $2^{n+1} = 2 \cdot 2^n$, so $2^{n+1} \leq c \cdot 2^n$ holds with $c = 2$ for every $n \geq 1$. Claim (b) is **true** (and in fact $2^{n+1} = \Theta(2^n)$, with $c_1 = c_2 = 2$). A constant factor in front of the exponent's *argument*, $2^n \to 2^{n+1}$, is only a constant factor in the value.

Solution (c). Suppose, for contradiction, that constants $c > 0$ and n_0 exist with $2^{2n} \leq c \cdot 2^n$ for all $n \geq n_0$. Since $2^{2n} = 2^n \cdot 2^n$, divide both sides by $2^n > 0$: $2^n \leq c$ for all $n \geq n_0$. But 2^n grows without bound. Take $n = \max(n_0, \lfloor \lg c \rfloor + 1)$; then $n \geq n_0$ and $n > \lg c$, so $2^n > c$ — a contradiction. Claim (c) is **false**. A constant factor in the *exponent*, $2^n \to 2^{2n} = 4^n$, is not a constant factor in the value; it changes the base.

Solution (d). *Upper bound.* $n! = 1 \cdot 2 \cdot 3 \cdots n$ is a product of n factors, none larger than n, so $n! \leq n^n$. Taking logarithms (which preserve order), $\lg(n!) \leq \lg(n^n) = n \lg n$. So $c_2 = 1$, for all $n \geq 1$.

Lower bound. Keep only the top half of the factors. The factors $\lceil n/2 \rceil, \ldots, n$ number $n - \lceil n/2 \rceil + 1 \geq n/2$, and each is at least $n/2$; the discarded factors are at least 1. So $n! \geq (n/2)^{n/2}$, and

$$\lg(n!) \geq \frac{n}{2} \lg \frac{n}{2} = \frac{n}{2}(\lg n - 1).$$

We want this to be at least $c_1 n \lg n$ with $c_1 = \frac{1}{4}$:

$$\frac{n}{2}(\lg n - 1) \geq \frac{n}{4} \lg n \Leftrightarrow 2(\lg n - 1) \geq \lg n \Leftrightarrow \lg n \geq 2 \Leftrightarrow n \geq 4.$$

So $c_1 = \frac{1}{4}, c_2 = 1, n_0 = 4$. Claim (d) is **true**.

Answer. (a) true, $c_1 = 4$, $c_2 = 5$, $n_0 = 4$. (b) true, $c = 2$, $n_0 = 1$. (c) false. (d) true, $c_1 = \frac{1}{4}$, $c_2 = 1$, $n_0 = 4$.

Program. A program cannot prove an asymptotic statement — it can only check finitely many n — but it is an excellent way to catch a wrong constant. **Listing 4.10** tests the constants of (a) and (d) for every n from 4 to 10^6, and for (c) it finds, for three values of c, the first n at which 4^n escapes $c \cdot 2^n$. It computes $\lg(n!)$ with `lgamma`, the logarithm of the gamma function, since $\Gamma(n + 1) = n!$ and $n!$ itself would overflow at $n = 21$.

Listing 4.10 Numerical checks of the witnesses c_1, c_2, n_0 for claims (a) and (d), and of the escape point in claim (c).

```
#include <cstdio>
#include <initializer_list>
#include <cmath>
using namespace std;
typedef long long LL;

int main() {
    // (a) 4n^2 <= 5n^2 - 3n + 12 <= 5n^2 for all n >= 4
    printf("(a)  n   4n^2   f(n)=5n^2-3n+12   5n^2   upper holds?\n");
    for (LL n = 1; n <= 6; ++n) {
        LL f = 5 * n * n - 3 * n + 12;
        printf("   %2lld  %5lld  %16lld  %5lld   %s\n", n, 4 * n * n, f, 5 * n * n,
               f <= 5 * n * n ? "yes" : "no");
    }
    LL bad = 0;
    for (LL n = 4; n <= 1000000; ++n) {
        LL f = 5 * n * n - 3 * n + 12;
        if (!(4 * n * n <= f && f <= 5 * n * n)) ++bad;
    }
    printf("     violations for 4 <= n <= 10^6: %lld\n", bad);

    // (c) 4^n <= c * 2^n fails as soon as 2^n > c
    printf("(c)  c          first n with 4^n > c*2^n\n");
    for (double c : {10.0, 1000.0, 1000000.0}) {
        int n = 0;
        while (pow(4.0, n) <= c * pow(2.0, n)) ++n;
        printf("     %-9.0f  %d\n", c, n);
    }

    // (d) (1/4) n lg n <= lg(n!) <= n lg n for n >= 4
    printf("(d)  n        lg(n!)      (1/4)n lg n     n lg n\n");
    for (double n : {4.0, 16.0, 1024.0, 1048576.0}) {
        double lgfact = lgamma(n + 1.0) / log(2.0);    // lg(n!) without computing n!
        printf("     %-8.0f %12.2f  %12.2f  %12.2f\n", n, lgfact, 0.25 * n * log2(n), n *
log2(n));
    }
    int badd = 0;
    for (int n = 4; n <= 1000000; ++n) {
        double lgfact = lgamma(n + 1.0) / log(2.0), nl = n * log2((double)n);
        if (!(0.25 * nl <= lgfact && lgfact <= nl)) ++badd;
    }
    printf("     violations for 4 <= n <= 10^6: %d\n", badd);
    return 0;
}
```

```
(a)  n   4n^2   f(n)=5n^2-3n+12   5n^2   upper holds?
    1      4                14      5   no
    2     16                26     20   no
    3     36                48     45   no
    4     64                80     80   yes
    5    100               122    125   yes
    6    144               174    180   yes
     violations for 4 <= n <= 10^6: 0
(c)  c          first n with 4^n > c*2^n
     10         4
     1000       10
     1000000    20
(d)  n        lg(n!)      (1/4)n lg n     n lg n
     4                4.58          2.00          8.00
     16              44.25         16.00         64.00
     1024          8769.01       2560.00      10240.00
     1048576   19458755.93    5242880.00   20971520.00
     violations for 4 <= n <= 10^6: 0
```

The first block shows the upper bound of (a) failing at $n = 1,2,3$ and holding from $n = 4$, exactly as the algebra said. The escape points in (c) are $\lfloor \lg c \rfloor + 1$: 4 for $c = 10$, 10 for $c = 1000$, 20 for $c = 10^6$ — whatever c you choose, 2^{2n} escapes.

> **CHECK YOURSELF.** Prove $(n+3)^2 = \Theta(n^2)$ with explicit constants.
>
> *Answer.* Lower: $(n+3)^2 \geq n^2$ for all $n \geq 1$, so $c_1 = 1$. Upper: for $n \geq 3$, $n + 3 \leq 2n$, so $(n+3)^2 \leq 4n^2$, and $c_2 = 4$. Hence $c_1 = 1, c_2 = 4, n_0 = 3$.

4.6.3 Worked Problem F — ranking thirteen functions by growth

Problem. Arrange these functions in increasing order of growth, grouping any that are Θ of each other:

$$100,\quad \lg n,\quad \lg^2 n,\quad \sqrt{n},\quad 2^{\lg n},\quad n\lg n,\quad \lg(n!),\quad n^2,\quad 4^{\lg n},\quad n^3,\quad (3/2)^n,\quad 2^n,\quad n!.$$

What it tests. The growth hierarchy of Section 2.12.4, the facts about logarithms, exponentials and factorials of Section 3.8, and the habit of simplifying a function before comparing it.

Solution. First *simplify* the two functions in disguise, using the identity $a^{\lg n} = n^{\lg a}$ (take lg of both sides: $\lg n \cdot \lg a$ either way):

- $2^{\lg n} = n^{\lg 2} = n^1 = n$.
- $4^{\lg n} = n^{\lg 4} = n^2$. So $4^{\lg n}$ and n^2 are not merely Θ of each other; they are equal.

Also, by Worked Problem E(d), $\lg(n!) = \Theta(n\lg n)$. Now order the list, justifying each step with a limit of the ratio: if $f(n)/g(n) \to 0$ then $f = o(g)$ and f comes strictly first.

1. 100 is a constant: $\Theta(1)$.
2. $\lg n \to \infty$, so the constant comes first.
3. $\frac{\lg n}{\lg^2 n} = \frac{1}{\lg n} \to 0$, so $\lg n$ precedes $\lg^2 n$.
4. $\lg^2 n = o(\sqrt{n})$: every power of a logarithm is beaten by every positive power of n (Section 3.8.6). Substituting $n = 2^m$ makes it vivid: $\lg^2 n = m^2$ while $\sqrt{n} = 2^{m/2}$, a polynomial against an exponential in m.
5. $\frac{\sqrt{n}}{n} = \frac{1}{\sqrt{n}} \to 0$, so $\sqrt{n}$ precedes $n = 2^{\lg n}$.
6. $\frac{n}{n\lg n} = \frac{1}{\lg n} \to 0$, so n precedes $n\lg n$, which ties with $\lg(n!)$.
7. $\frac{n\lg n}{n^2} = \frac{\lg n}{n} \to 0$, so next come n^2 and its twin $4^{\lg n}$.
8. $\frac{n^2}{n^3} = \frac{1}{n} \to 0$: then n^3.
9. $n^3 = o((3/2)^n)$: every polynomial is beaten by every exponential with base greater than 1 (Section 3.8.5).
10. $\frac{(3/2)^n}{2^n} = \left(\frac{3}{4}\right)^n \to 0$: a smaller base loses.
11. $\frac{n!}{2^n} = \frac{1}{2} \cdot \frac{2}{2} \cdot \frac{3}{2} \cdots \frac{n}{2} \geq \frac{1}{2}\left(\frac{3}{2}\right)^{n-2} \to \infty$ for $n \geq 2$, since the first two factors multiply to $\frac{1}{2}$ and each of the remaining $n - 2$ is at least $\frac{3}{2}$. So 2^n precedes $n!$.

Table 4.8 records the ranking.

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 $\geq \frac{1}{2}(3/2)^{n-2} \to \infty$

Answer. $100 \prec \lg n \prec \lg^2 n \prec \sqrt{n} \prec 2^{\lg n} \prec n\lg n \asymp \lg(n!) \prec n^2 = 4^{\lg n} \prec n^3 \prec (3/2)^n \prec 2^n \prec n!$, where $\prec$ means "is little-o of" and $\asymp$ means "is Θ of".

Program. An asymptotic ranking describes *large* n. What happens at the sizes you actually meet? **Listing 4.11** prints $\lg f(n)$ — the number of binary digits of $f(n)$ — for each function at $n = 2^4, 2^8, 2^{16}, 2^{32}$. Printing the logarithm keeps even $n!$ on the page.

Listing 4.11 The binary logarithm of each of the thirteen functions at four sizes, showing where the asymptotic order does and does not yet hold.

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

// Prints lg f(n) -- the number of binary digits of f(n) -- so that even n! fits on the
page.
int main() {
    const char* name[] = {"100", "lg n", "lg^2 n", "sqrt n", "2^(lg n)", "n lg n",
"lg(n!)",
                          "n^2", "4^(lg n)", "n^3", "(3/2)^n", "2^n", "n!"};
    double ns[] = {16.0, 256.0, 65536.0, 4294967296.0};
    printf("%-10s", "f(n)");
    for (double n : ns) printf("   n=2^%-6.0f", log2(n));
    printf("\n");
    for (int f = 0; f < 13; ++f) {
        printf("%-10s", name[f]);
        for (double n : ns) {
            double L = log2(n), v;                          // v = lg f(n)
            switch (f) {
                case 0:  v = log2(100.0); break;
                case 1:  v = log2(L); break;
                case 2:  v = 2 * log2(L); break;
                case 3:  v = L / 2; break;
                case 4:  v = L; break;                      // 2^(lg n) = n
                case 5:  v = L + log2(L); break;
                case 6:  v = log2(lgamma(n + 1) / log(2.0)); break;
                case 7:  v = 2 * L; break;
                case 8:  v = 2 * L; break;                  // 4^(lg n) = n^2
                case 9:  v = 3 * L; break;
                case 10: v = n * log2(1.5); break;
                case 11: v = n; break;
                default: v = lgamma(n + 1) / log(2.0); break;
            }
            printf("   %12.4g", v);
        }
        printf("\n");
    }
    return 0;
}
```

```
f(n)         n=2^4        n=2^8        n=2^16       n=2^32
100                6.644        6.644        6.644        6.644
lg n                   2            3            4            5
lg^2 n                 4            6            8           10
sqrt n                 2            4            8           16
2^(lg n)               4            8           16           32
n lg n                 6           11           20           37
lg(n!)             5.468        10.72        19.86        36.93
n^2                    8           16           32           64
4^(lg n)               8           16           32           64
n^3                   12           24           48           96
(3/2)^n            9.359        149.8     3.834e+04    2.512e+09
2^n                   16          256     6.554e+04    4.295e+09
n!                44.25         1684      9.54e+05    1.312e+11
```

Read the columns, not just the rows. At $n = 16$ the "slowest" function, the constant $100 \approx 2^{6.64}$, is larger than $\lg^2 n = 2^4$, larger than $\sqrt{n} = 2^2$, and even larger than $n = 2^4$ itself; and $(3/2)^{16} \approx 2^{9.36}$ is still smaller than $n^3 = 2^{12}$. At $n = 2^{16}$, $\lg^2 n$ and $\sqrt{n}$ are exactly equal, both $2^8 = 256$ — before that point the logarithm squared is the larger. Only by $n = 2^{32}$ is every row larger than the one above it. The asymptotic order is a statement about the long run, and the long run can start later than you think; this is why Chapter 3 insisted on the threshold n_0.

> **CHECK YOURSELF.** Where do $n^{1/\lg n}$ and $n^{\lg 7}$ fit in the ranking?

Answer. $n^{1/\lg n} = (2^{\lg n})^{1/\lg n} = 2^1 = 2$, a constant: it joins row 1 with 100. And $\lg 7 \approx 2.807$, so $n^{\lg 7}$ sits strictly between n^2 and n^3 — it is the running time of Strassen's matrix multiplication, which Chapter 3 met as the recurrence $T(n) = 7T(n/2) + n^2$.

4.7 Worked Problems on Recurrences

Chapter 3 gave three methods for solving a recurrence: substitution, the recursion tree, and the Master Theorem. The three worked problems of this section use one method each, and then cross-check the answer with a second method and with a program that evaluates the recurrence exactly. That habit — solve one way, confirm another way — is the most reliable protection against the small slips that recurrences invite.

4.7.1 Worked Problem G — the substitution method, and a hypothesis that must be strengthened

Problem. (a) Prove that $T(n) = T(\lfloor n/2 \rfloor) + n$ with $T(1) = 1$ is $\Theta(n)$. (b) Prove that

$$U(n) = 2\,U(\lfloor n/2 \rfloor) + 1, \qquad U(1) = 1 \tag{4.11}$$

is $O(n)$, and explain why the most natural induction hypothesis fails.

What it tests. The substitution method of Section 3.11: guess the form, prove it by induction, handle the base case, and — when the induction almost works — strengthen the hypothesis rather than weaken the claim.

Solution (a). *Upper bound: guess $T(n) \le 2n$.* Base case: $T(1) = 1 \le 2 \cdot 1$. ✓ Inductive step: let $n \ge 2$ and assume $T(m) \le 2m$ for every $m < n$. Since $\lfloor n/2 \rfloor < n$,

$$T(n) = T(\lfloor n/2 \rfloor) + n \le 2\lfloor n/2 \rfloor + n \le 2 \cdot \frac{n}{2} + n = 2n. \ \checkmark$$

Lower bound: $T(n) \ge n$. Every value of T is positive (a positive number plus a positive n), so $T(n) = T(\lfloor n/2 \rfloor) + n \ge n$ for $n \ge 2$, and $T(1) = 1 \ge 1$. ✓

So $n \le T(n) \le 2n$ for every $n \ge 1$: $T(n) = \Theta(n)$ with $c_1 = 1$, $c_2 = 2$, $n_0 = 1$. The intuition is the geometric series of Section 3.9.3: $n + n/2 + n/4 + \cdots < 2n$.

Solution (b). *The natural guess, $U(n) \le cn$, fails.* Substitute the guess into Equation (4.11). Assume $U(m) \le cm$ for every $m < n$. Then

$$U(n) = 2U(\lfloor n/2 \rfloor) + 1 \le 2c\lfloor n/2 \rfloor + 1 \le cn + 1.$$

We needed $U(n) \le cn$, and we got $cn + 1$. No choice of c rescues this: $cn + 1 \le cn$ is false for every c. It is tempting to shrug — "$cn + 1$ is still $O(n)$" — but that is exactly the error Section 3.11.3 warned against. The induction must prove *the same inequality with the same constant* that it assumed; otherwise the constant drifts upward by one at every level and proves nothing.

The fix: subtract a lower-order term. Guess instead $U(n) \le cn - d$ for constants c and $d \ge 0$ to be chosen. Assume it for all $m < n$:

$$U(n) \le 2(c\lfloor n/2 \rfloor - d) + 1 \le cn - 2d + 1.$$

We need $cn - 2d + 1 \leq cn - d$, which simplifies to $d \geq 1$. Take $d = 1$. The base case requires $U(1) = 1 \leq c \cdot 1 - 1$, that is $c \geq 2$. Take $c = 2$. So $U(n) \leq 2n - 1$ for all $n \geq 1$, and $U(n) = O(n)$. ✓

The paradox is worth stating plainly: the weaker-looking claim $U(n) \leq 2n$ is *true*, yet it cannot be proved by this induction, while the stronger claim $U(n) \leq 2n - 1$ can. A stronger hypothesis gives the inductive step more to work with. For the matching lower bound, $U(n) \geq 2\lfloor n/2 \rfloor + 1 \geq (n - 1) + 1 = n$ by the same argument as in (a), so $U(n) = \Theta(n)$. (In fact $U(n) = 2^{\lfloor \lg n \rfloor + 1} - 1$ exactly: the number of nodes in a complete binary tree with $\lfloor \lg n \rfloor + 1$ levels.)

Answer. (a) $n \leq T(n) \leq 2n$, so $T(n) = \Theta(n)$. (b) $n \leq U(n) \leq 2n - 1$, so $U(n) = \Theta(n)$; the proof needs the strengthened hypothesis $U(n) \leq cn - 1$.

Program. **Listing 4.12** tabulates both recurrences for $n \leq 10$ and checks the proved bounds for every n up to 10^6; it also checks the naive claim $U(n) \leq n$, to show how quickly a wrong constant is caught.

Listing 4.12 Both recurrences of Worked Problem G evaluated exactly, with the proved bounds checked for every $n \leq 10^6$.

```
#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
```

The U column is 1,3,3,7,7,7,7,15, ...: it meets $2n-1$ exactly at the powers of two, so the constant $c=2$ cannot be improved. And the claim with $c=1$ fails immediately, at $n=2$, where $U(2)=3>2$.

CHECK YOURSELF. Prove that $T(n)=T(\lfloor n/2\rfloor)+1$ with $T(1)=1$ satisfies $T(n)\le \lg n+1$.

Answer. Base: $T(1)=1=\lg 1+1$. Step, for $n\ge 2$: $T(n)\le \lg\lfloor n/2\rfloor+1+1\le \lg(n/2)+2=\lg n-1+2=\lg n+1$. This is the recurrence of binary search, and the bound is the $\lfloor \lg n\rfloor+1$ probes of Worked Problem C.

4.7.2 Worked Problem H — a recursion tree with a table of levels

Problem. Solve

$$T(n)=3T(n/4)+n^2,\qquad T(1)=1 \tag{4.12}$$

for n a power of 4, by drawing its recursion tree. Find the exact value of $T(n)$ and its Θ class, then confirm the bound by substitution and by the Master Theorem.

What it tests. The recursion-tree method of Section 3.12: count the nodes and the cost at each level, sum the levels (usually a geometric series), and add the leaves.

Solution. Each call of Equation (4.12) does n^2 work of its own and makes three calls on inputs a quarter the size. Write $n=4^k$, so the tree has levels $0,1,\dots,k$, with $k=\log_4 n$. The root costs n^2 and has three children, each a problem of size $n/4$. Each of those costs $(n/4)^2$ and has three children of size $n/16$, and so on. The drawing below shows the top of the tree.

```
level 0                          n^2                                total  n^2
                       /          |          \
level 1          (n/4)^2      (n/4)^2      (n/4)^2                  total  (3/16) n^2
                 /  |  \      /  |  \      /  |  \
level 2     nine nodes, each (n/16)^2                               total  (3/16)^2 n^2
              ...
level i     3^i nodes, each (n/4^i)^2                               total  (3/16)^i n^2
              ...
level k     3^k = n^(log_4 3) leaves, each T(1) = 1                 total  n^(log_4 3)
```

The level cost is the number of nodes times the cost per node. At level $i<k$ there are 3^i nodes of size $n/4^i$, each costing $(n/4^i)^2=n^2/16^i$, so the level costs

$$3^i\cdot\frac{n^2}{16^i}=\left(\frac{3}{16}\right)^i n^2.$$

At level k there are 3^k leaves, each costing $T(1) = 1$. Since $3^k = 3^{\log_4 n} = n^{\log_4 3} \approx n^{0.792}$, the leaves cost $n^{\log_4 3}$ in total. **Table 4.9** is the table of levels.

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	$n^{\log_4 3}$

Summing the levels. The internal levels form a geometric series with ratio $r = \frac{3}{16}$, which Section 3.9.3 sums as $\sum_{i=0}^{k-1} r^i = \frac{1-r^k}{1-r}$. Here $1 - r = \frac{13}{16}$, and $r^k = \frac{3^k}{16^k} = \frac{3^k}{n^2}$, because $16^k = (4^k)^2 = n^2$. So the internal levels cost

$$n^2 \cdot \frac{1 - 3^k/n^2}{13/16} = \frac{16}{13}(n^2 - 3^k),$$

and adding the leaves,

$$T(n) = \frac{16}{13}n^2 - \frac{16}{13}3^k + 3^k = \frac{16}{13}n^2 - \frac{3}{13}3^k.$$

$$T(n) = \frac{16\,n^2 - 3^{k+1}}{13}, \qquad n = 4^k \tag{4.13}$$

Checking Equation (4.13) *by hand.* $n = 4$, $k = 1$: the recurrence gives $T(4) = 3T(1) + 16 = 3 + 16 = 19$; the formula gives $\frac{16 \cdot 16 - 9}{13} = \frac{256-9}{13} = \frac{247}{13} = 19$ ✓. $n = 16$, $k = 2$: $T(16) = 3 \cdot 19 + 256 = 313$; the formula gives $\frac{16 \cdot 256 - 27}{13} = \frac{4096-27}{13} = \frac{4069}{13} = 313$ ✓.

The Θ class. The root alone costs n^2, so $T(n) \geq n^2$; and Equation (4.13) gives $T(n) < \frac{16}{13}n^2$. Hence $T(n) = \Theta(n^2)$, and the ratio $T(n)/n^2$ must approach $\frac{16}{13} \approx 1.230769$.

Confirmation by substitution. The tree suggested the bound $T(n) \leq \frac{16}{13}n^2$; the substitution method proves it without any summation. Base: $T(1) = 1 \leq \frac{16}{13}$. ✓ Step: assuming the bound at $n/4$,

$$T(n) = 3T(n/4) + n^2 \leq 3 \cdot \frac{16}{13} \cdot \frac{n^2}{16} + n^2 = \frac{3}{13}n^2 + \frac{13}{13}n^2 = \frac{16}{13}n^2. \ ✓$$

This is the workflow Chapter 3 recommended: the recursion tree *finds* the answer, the substitution method *proves* it.

Confirmation by the Master Theorem. $a = 3$, $b = 4$, $n^{\log_b a} = n^{\log_4 3} \approx n^{0.792}$, and $f(n) = n^2 = \Omega(n^{0.792+\varepsilon})$ with $\varepsilon \approx 1.2$. Regularity: $af(n/b) = 3(n/4)^2 = \frac{3}{16}n^2$, so $c = \frac{3}{16} < 1$. Case 3 applies: $T(n) = \Theta(n^2)$. ✓ The regularity constant $\frac{3}{16}$ is exactly the ratio between consecutive levels of the tree — which is what the regularity condition means.

Answer. $T(n) = \frac{16n^2 - 3^{k+1}}{13}$ for $n = 4^k$, so $T(n) = \Theta(n^2)$ and $T(n)/n^2 \to \frac{16}{13}$.

Program. **Listing 4.13** prints the level table for $n = 256$ and then the ratio $T(n)/n^2$ for $n = 4, 16, \ldots, 4^{12}$.

Listing 4.13 The recursion tree of $T(n) = 3T(n/4) + n^2$ at $n = 256$, level by level, and the ratio $T(n)/n^2$ as n grows.

```
#include <cstdio>
#include <cmath>
using namespace std;
typedef long long LL;

// T(n) = 3T(n/4) + n^2 with T(1) = 1, for n a power of 4.
LL T(LL n) { return n == 1 ? 1 : 3 * T(n / 4) + n * n; }

int main() {
    LL n = 256;
    printf("Recursion tree for n = %lld\n", n);
    printf("level  nodes  size  cost/node  level total  total/n^2\n");
    LL nodes = 1, size = n, sum = 0;
    for (int i = 0; size >= 1; ++i) {
        LL per = (size == 1) ? 1 : size * size;   // a leaf costs T(1) = 1
        LL tot = nodes * per;
        sum += tot;
        printf("%5d  %5lld  %4lld  %9lld  %11lld  %9.6f\n", i, nodes, size, per, tot,
(double)tot / (n * n));
        if (size == 1) break;
        nodes *= 3; size /= 4;
    }
    printf("sum of the levels = %lld, T(256) computed directly = %lld\n\n", sum, T(n));
    printf("        n            T(n)   T(n)/n^2\n");
    for (LL m = 4, k = 1; k <= 12; m *= 4, ++k)
        printf("%9lld  %14lld   %.6f\n", m, T(m), (double)T(m) / ((double)m * m));
    printf("16/13 = %.6f\n", 16.0 / 13.0);
    return 0;
}
```

```
Recursion tree for n = 256
level  nodes  size  cost/node  level total  total/n^2
    0      1   256      65536        65536   1.000000
    1      3    64       4096        12288   0.187500
    2      9    16        256         2304   0.035156
    3     27     4         16          432   0.006592
    4     81     1          1           81   0.001236
sum of the levels = 80641, T(256) computed directly = 80641

        n            T(n)   T(n)/n^2
        4              19   1.187500
       16             313   1.222656
       64            5035   1.229248
      256           80641   1.230484
     1024         1290499   1.230716
     4096        20648713   1.230759
    16384       330381595   1.230767
    65536      5286112081   1.230769
   262144     84577812979   1.230769
  1048576   1353245066713   1.230769
  4194304  21651921244555   1.230769
 16777216  346430740444321   1.230769
16/13 = 1.230769
```

The level totals 65536,12288,2304,432 shrink by the factor $\frac{3}{16} = 0.1875$ each time, the leaves add $81 = 3^4$, and the sum 80,641 is exactly $\frac{16 \cdot 65536 - 3^5}{13} = \frac{1{,}048{,}576 - 243}{13} = \frac{1{,}048{,}333}{13}$. The ratio column climbs to $1.230769 = \frac{16}{13}$ and stays there.

> **CHECK YOURSELF.** Draw the recursion tree of $T(n) = 4T(n/2) + n$ and find its Θ class.
>
> *Answer.* Level i has 4^i nodes of size $n/2^i$, costing $4^i \cdot n/2^i = 2^i n$: the levels *grow* by a factor of 2. The leaves, at level $\lg n$, number $4^{\lg n} = n^2$ and dominate; the internal levels sum to $n(2^{\lg n} - 1) = n^2 - n$. So $T(n) = \Theta(n^2)$ — Master Theorem Case 1, with $n^{\log_2 4} = n^2$.

4.7.3 Worked Problem I — four recurrences and the Master Theorem, including a gap

Problem. Use the Master Theorem, where it applies, to solve

(a) $T(n) = 8T(n/4) + n$; (b) $T(n) = 27T(n/3) + n^3$; (c) $T(n) = 5T(n/4) + n^2$; (d) $T(n) = T(n/2) + \lg n$,

each with $T(1) = 1$. Where the theorem does not apply, say exactly why and solve the recurrence another way.

What it tests. The statement of the Master Theorem in Section 3.13.1, the computation of $\log_b a$, the choice of ε, the regularity condition, and the gaps of Section 3.13.6.

Solution. For each recurrence, follow the five-step strategy of Section 3.13.7: read off a, b and $f(n)$; compute the critical exponent $\log_b a$; compare $f(n)$ with $n^{\log_b a}$; check the side condition; conclude.

(a) $a = 8$, $b = 4$, $f(n) = n$. $\log_4 8 = \frac{\lg 8}{\lg 4} = \frac{3}{2}$, so $n^{\log_b a} = n^{1.5}$. The exponent of f is $1 < 1.5$, so $f(n) = O(n^{1.5-\varepsilon})$ with $\varepsilon = 0.5$. **Case 1**: $T(n) = \Theta(n^{1.5})$. The leaves dominate.

(b) $a = 27$, $b = 3$, $f(n) = n^3$. $\log_3 27 = 3$, so $n^{\log_b a} = n^3$, and $f(n) = \Theta(n^3)$ exactly. **Case 2**: $T(n) = \Theta(n^3 \lg n)$. Every level costs the same n^3, and there are $\log_3 n + 1$ levels.

(c) $a = 5,\ b = 4,\ f(n) = n^2$. $\log_4 5 = \frac{\lg 5}{\lg 4} \approx \frac{2.3219}{2} \approx 1.161$. The exponent of f is $2 > 1.161$, so $f(n) = \Omega(n^{1.161+\varepsilon})$ with $\varepsilon \approx 0.839$. Regularity: $af(n/b) = 5(n/4)^2 = \frac{5}{16}n^2$, so $c = \frac{5}{16} < 1$. ✓ **Case 3**: $T(n) = \Theta(n^2)$. The root dominates.

(d) $a = 1, b = 2, f(n) = \lg n$. $\log_2 1 = 0$, so $n^{\log_b a} = n^0 = 1$. Now test the three cases.

- Case 1 needs $\lg n = O(n^{-\varepsilon})$ for some $\varepsilon > 0$. But $n^{-\varepsilon} \to 0$ while $\lg n \to \infty$. No.
- Case 2 needs $\lg n = \Theta(1)$. But $\lg n$ is unbounded. No.
- Case 3 needs $\lg n = \Omega(n^{\varepsilon})$ for some $\varepsilon > 0$. But $\lg n = o(n^{\varepsilon})$ for every $\varepsilon > 0$ (Section 3.8.6). No.

$f(n)$ is larger than n^0, but not *polynomially* larger. The recurrence falls into the gap between Cases 2 and 3, and **the Master Theorem does not apply**. We solve it by unrolling instead. With $n = 2^k$, $\lg n = k$, and each unrolling step halves n and lowers the logarithm by one:

$$\begin{aligned} T(2^k) = k + T(2^{k-1}) = k + (k-1) + T(2^{k-2}) &= \cdots \\ &= k + (k-1) + \cdots + 1 + T(1). \end{aligned}$$

That is the arithmetic series once more:

$$T(n) = \frac{k(k+1)}{2} + 1 = \frac{\lg n\,(\lg n + 1)}{2} + 1 = \Theta(\lg^2 n).$$

Check at $n = 16, k = 4$: $T(16) = 4 + 3 + 2 + 1 + 1 = 11$, and $\frac{4 \cdot 5}{2} + 1 = 11$ ✓.

Table 4.10 gathers the four results, together with the exact values on powers of b that a short calculation (or **Listing 4.14**) confirms.

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	$\frac{\lg n(\lg n + 1)}{2} + 1$

Other ways the theorem can fail to apply. The gap is not the only one. The theorem also says nothing when the recurrence is not of the form $aT(n/b) + f(n)$ with constants $a \geq 1$, $b > 1$: for example $T(n) = 2T(n-1) + 1$ (the argument is $n - 1$, not n/b; unrolling gives $\Theta(2^n)$), $T(n) = \frac{1}{2}T(n/2) + n$ (fewer than one subproblem, $a < 1$), or $T(n) = 2^n T(n/2) + n$ (the number of subproblems depends on n). Before reaching for the theorem, check that it is allowed.

Answer. (a) $\Theta(n^{1.5})$; (b) $\Theta(n^3 \lg n)$; (c) $\Theta(n^2)$; (d) the Master Theorem does not apply, and $T(n) = \frac{\lg n(\lg n+1)}{2} + 1 = \Theta(\lg^2 n)$.

Program. **Listing 4.14** evaluates each recurrence exactly on powers of b in `long double` and divides by the predicted bound. A ratio that settles to a constant confirms the Θ class.

Listing 4.14 The four recurrences of Worked Problem I evaluated exactly and divided by their predicted bounds.

```
#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
```

Each ratio settles where the exact formulas of **Table 4.10** say it must: (a) towards 2; (b) towards 1, slowly, as $1 + \frac{1}{\log_3 n}$; (c) towards $\frac{16}{11} \approx 1.454545$; (d) towards $\frac{1}{2}$, as $\frac{k(k+1)/2+1}{k^2}$. Notice how slowly (b) and (d) converge. A ratio that drifts like $1 + 1/k$ is a warning that you are looking at a logarithmic factor, and it is easy to misread such a column as "not settling" if you do not know what to expect.

> **CHECK YOURSELF.** Does the Master Theorem apply to $T(n) = 9T(n/3) + n^2 \lg n$? If not, solve it.
>
> *Answer.* $n^{\log_3 9} = n^2$, and $f(n) = n^2 \lg n$ is larger than n^2 but only by a logarithmic factor, not by n^ε: the gap again. In the recursion tree, level i has 9^i nodes of size $n/3^i$, costing $9^i (n/3^i)^2 \lg(n/$

$3^i) = n^2(\lg n - i\lg 3)$. Summing over the $\log_3 n$ levels gives n^2 times an arithmetic series of about $\log_3 n$ terms averaging $\frac{1}{2}\lg n$, so $T(n) = \Theta(n^2 \lg^2 n)$.

4.8 Worked Problems on Closed Forms and Hidden Recurrences

The two problems of this section are written in the style of an online-judge problem, set in the Profitina Laundry case study that runs through the book. The first is a cousin of TOHU: a sum that telescopes. The second hides the Fibonacci numbers of Section 3.8.9 inside a classic counting question about domino tilings [5], and then asks for them at an index far too large for any table.

4.8.1 Worked Problem J — Laundry Tokens

Problem.

LAUNDRT — Laundry Tokens. Profitina Laundry hands out one token on day 1, two on day 2, three on day 3, and so on, so that $T_k = 1 + 2 + \cdots + k$ tokens are in circulation by the end of day k. Each day it also pays a loyalty bonus to the whole pool of token holders, and the bonus paid on day k is $1/T_k$: the more tokens there are, the smaller the day's bonus. Given n, compute the total bonus paid on days 1 through n, that is

$$V_n = \sum_{k=1}^{n} \frac{1}{T_k}.$$

Input: $T \leq 20000$ test cases, one integer n each, $1 \leq n \leq 10^9$. Output: V_n with 11 digits after the decimal point. Target complexity: $O(1)$.

What it tests. The triangular numbers of Equation (4.1), a telescoping sum (Section 3.9.5), and the arithmetic traps of printing a ratio.

Solution. *Steps 1–3.* Given: n up to 10^9, up to 20,000 times. Wanted: $\sum_{k=1}^{n} 1/T_k$, to eleven decimals. The budget, like the note $O(1)$, forbids adding up 10^9 terms twenty thousand times: find a closed form.

Step 4 — the first cases by hand.

- $T_1 = 1$, so the first term is 1. $V_1 = 1$.
- $T_2 = 3$, term $1/3$. $V_2 = 1 + 1/3 = 4/3$.
- $T_3 = 6$, term $1/6$. $V_3 = 4/3 + 1/6 = \frac{8}{6} + \frac{1}{6} = 9/6 = 3/2$.
- $T_4 = 10$, term $1/10$. $V_4 = 3/2 + 1/10 = \frac{15}{10} + \frac{1}{10} = \frac{16}{10} = 8/5$.
- $T_5 = 15$, term $1/15$. $V_5 = 8/5 + 1/15 = \frac{24}{15} + \frac{1}{15} = \frac{25}{15} = 5/3$.

The partial sums are $1/1, 4/3, 3/2, 8/5, 5/3$. Rewrite each with denominator $n+1$: $\frac{2}{2}, \frac{4}{3}, \frac{6}{4}, \frac{8}{5}, \frac{10}{6}$. The pattern is unmistakable: $V_n = \frac{2n}{n+1}$.

Steps 5–6 — prove it. The shape is *closed form from a sum*. Since $T_k = \frac{k(k+1)}{2}$,

$$\frac{1}{T_k} = \frac{2}{k(k+1)} = 2\left(\frac{1}{k} - \frac{1}{k+1}\right),$$

because $\frac{1}{k} - \frac{1}{k+1} = \frac{(k+1)-k}{k(k+1)} = \frac{1}{k(k+1)}$. Summing, every interior term cancels against its neighbor:

$$V_n = 2\sum_{k=1}^{n}\left(\frac{1}{k} - \frac{1}{k+1}\right) = 2\left(1 - \frac{1}{n+1}\right) = \frac{2n}{n+1}$$

(4.14)

Step 7 — the program. **Pseudocode 4.7** is the whole algorithm.

Pseudocode 4.7 The Laundry Tokens algorithm: one evaluation of Equation (4.14) per query.

```
LAUNDRT()
read T
for q = 1 to T
    read n
    print 2n / (n + 1) with 11 decimal places
```

Cost: one multiplication and one division per test case, $\Theta(1)$ time and $O(1)$ space, $\Theta(T)$ overall. Correctness follows from the telescoping identity, which is exact. Accuracy: $V_n < 2$ always, and eleven decimals of a number below 2 need twelve significant digits; `double` supplies about sixteen. The closed form also tells the laundry's accountant something useful: $V_n \to 2$ as $n \to \infty$, so the total bonus is bounded however long the promotion runs.

Answer. $V_n = \frac{2n}{n+1}$.

Program. **Listing 4.15** is the program.

Listing 4.15 LAUNDRT — Laundry Tokens, solved in $O(1)$ by telescoping.

```
#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;
}
```

For $n = 1,2,3,10,10^9$ it prints

```
1.00000000000
1.33333333333
1.50000000000
1.81818181818
1.99999999800
```

and a program that adds the terms one at a time agrees to all eleven places for $n = 1$ through 12. The trap here is writing `2 * n / (n+1)` in integer arithmetic, which truncates to 1 for every $n \geq 1$; the conversion to `double` must happen *before* the division.

CHECK YOURSELF. Find a closed form for $\sum_{k=1}^{n} \frac{1}{k(k+2)}$.

Answer. $\frac{1}{k(k+2)} = \frac{1}{2}\left(\frac{1}{k} - \frac{1}{k+2}\right)$. The telescope now cancels terms two apart, so two terms survive at each end: $\frac{1}{2}\left(1 + \frac{1}{2} - \frac{1}{n+1} - \frac{1}{n+2}\right) = \frac{3}{4} - \frac{2n+3}{2(n+1)(n+2)}$. Check $n = 1$: $\frac{3}{4} - \frac{5}{12} = \frac{9-5}{12} = \frac{1}{3} = \frac{1}{1\cdot 3}$ ✓.

4.8.2 Worked Problem K — tiling a $2 \times N$ strip

Problem.

> **STRIPTILE — Tiling a $2 \times N$ Strip.** A corridor in the laundry is 2 tiles wide and N tiles long. It is to be covered completely by 1×2 tiles, each laid either horizontally or vertically, with no overlaps and no gaps. In how many ways can this be done? Report the answer modulo $10^9 + 7$. Input: $T \leq 10^5$ test cases, one integer N each, $0 \leq N \leq 10^{18}$. Output: one number per test case. Target complexity: $O(\log N)$.

What it tests. Deriving a recurrence from a counting argument, recognizing the Fibonacci numbers (Section 3.8.9), and evaluating a linear recurrence at an astronomically large index by divide and conquer.

Solution. *Steps 1–3.* Given: N up to 10^{18}, up to 10^5 times. Wanted: the number of complete tilings of a $2 \times N$ rectangle by dominoes, modulo $10^9 + 7$. A table of 10^{18} entries is out of the question; the target $O(\log N)$ says: a recurrence plus a halving trick.

Step 4 — count small cases by hand. Let D_N be the answer. Write V for a vertical tile and HH for a pair of horizontal tiles stacked on top of each other.

- $N = 0$: the empty corridor is tiled in exactly one way — by laying nothing. $D_0 = 1$.
- $N = 1$: a 2×1 strip takes exactly one vertical tile. $D_1 = 1$.
- $N = 2$: VV or HH. $D_2 = 2$.
- $N = 3$: VVV, VHH, HHV. $D_3 = 3$.
- $N = 4$: VVVV, VVHH, VHHV, HHVV, HHHH. $D_4 = 5$.

Steps 5–6 — the recurrence, argued rather than guessed. Look at the leftmost column. Exactly two things can happen.

- It is covered by one **vertical** tile. What remains is a $2 \times (N - 1)$ strip, tiled in D_{N-1} ways.
- It is covered by the left halves of two **horizontal** tiles, one in each row. (A single horizontal tile cannot cover it alone, because the square below or above it would then have to be covered by a vertical tile that sticks out of the column.) Those two tiles also occupy column 2, so column 2 is now full, and what remains is a $2 \times (N - 2)$ strip, tiled in D_{N-2} ways.

No other case exists, and the two cases are disjoint, so

$$D_N = D_{N-1} + D_{N-2}, \qquad D_0 = D_1 = 1 \tag{4.15}$$

That is the Fibonacci recurrence with shifted indices: $D_N = F_{N+1}$, where $F_0 = 0$, $F_1 = 1$. Check: $F_1, \ldots, F_5 = 1,1,2,3,5$ ✓.

Designing an $O(\log N)$ evaluation. Two roads lead to F_{N+1} mod p in $\Theta(\log N)$ steps.

The *characteristic equation* $r^2 - r - 1 = 0$ has roots $\varphi = \frac{1+\sqrt{5}}{2}$ and $\psi = \frac{1-\sqrt{5}}{2}$, giving Binet's formula $F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}$. It is beautiful, but modulo p it needs a square root of 5, which exists only because 5 happens to be a quadratic residue modulo $10^9 + 7$; the extra machinery is not worth it here.

Fast doubling needs only integer arithmetic. Start from the matrix identity

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \qquad (n \geq 1),$$

which holds by induction on n: for $n = 1$ both sides are $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, and multiplying the right side by $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ gives $\begin{pmatrix} F_{n+1} + F_n & F_{n+1} \\ F_n + F_{n-1} & F_n \end{pmatrix} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix}$. Now square the matrix for $n = k$ to get the matrix for $n = 2k$. The top-left entry of the square is $F_{k+1}^2 + F_k^2$, and it must equal F_{2k+1}. The top-right entry is $F_{k+1}F_k + F_kF_{k-1} = F_k(F_{k+1} + F_{k-1})$, and it must equal F_{2k}; using $F_{k-1} = F_{k+1} - F_k$ to eliminate F_{k-1} gives

$$F_{2k} = F_k(2F_{k+1} - F_k), \qquad F_{2k+1} = F_k^2 + F_{k+1}^2 \tag{4.16}$$

So from the pair (F_k, F_{k+1}) we get (F_{2k}, F_{2k+1}) in a constant number of multiplications, and one more addition gives F_{2k+2} when we need the odd step. A recursion that halves n at every call and returns the pair (F_n, F_{n+1}) therefore costs $\Theta(\log N)$ — the recurrence $T(n) = T(n/2) + \Theta(1)$ of binary search. **Pseudocode 4.8** is the procedure.

Pseudocode 4.8 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))
```

A trace by hand. The answer for $N = 5$ is $D_5 = F_6$, so we call FIB(6). **Table 4.11** follows the recursion from the bottom up.

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)	$0 \cdot (2-0) = 0$	$0 + 1 = 1$	odd: $(1,0+1) = (1,1)$
FIB(3)	1	(1,1)	$1 \cdot (2-1) = 1$	$1 + 1 = 2$	odd: $(2,1+2) = (2,3)$
FIB(6)	3	(2,3)	$2 \cdot (6-2) = 8$	$4 + 9 = 13$	even: (8,13)

So $D_5 = F_6 = 8$, matching the value $D_5 = D_4 + D_3 = 5 + 3 = 8$ from Equation (4.15).

Analysis. The depth of the recursion is $\lfloor \log_2 N \rfloor + 2 \leq 61$ for $N \leq 10^{18}$, with constant work per level, so each query costs $\Theta(\log N)$ time and $\Theta(\log N)$ stack space. In total $\Theta(T \log N) \approx 10^5 \times 61 \approx 6 \times 10^6$ operations — comfortably inside the budget of **Table 4.1**. Correctness of the recurrence is the two-case argument above; correctness of fast doubling follows from Equation (4.16) by induction on n.

Answer. $D_N = F_{N+1}$, computed modulo $10^9 + 7$ by fast doubling in $\Theta(\log N)$.

Program. **Listing 4.16** is the program. Two details deserve attention. First, every product is reduced modulo p immediately: both factors are below $p \approx 10^9$, so each product is below 10^{18} and fits in a `long long`, but the product of three such numbers would not. Second, the subtraction in $2F_{k+1} - F_k$ can go negative after reduction, and in C++ the `%` operator keeps the sign of the

dividend, so the code adds `MOD` before taking the remainder: `(2 * b % MOD - a % MOD + MOD) % MOD`.

Listing 4.16 STRIPTILE — tilings of a $2 \times N$ strip by fast doubling, $\Theta(\log N)$ per query.

```
#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;
}
```

For $N = 0,1,2,3,4,5,30, 10^{18}$ it prints

```
1
1
2
3
5
8
1346269
680057396
```

and a straightforward table built from Equation (4.15) confirms $D_0, \ldots, D_6 = 1,1,2,3,5,8,13$ and $D_{30} = 1{,}346{,}269$. The traps: answering F_N instead of F_{N+1} (check $N = 2$, which must give 2); forgetting that $D_0 = 1$, since an empty tiling is a tiling; the negative intermediate in $2F_{k+1} - F_k$; and trying to build a table of size N — 10^{18} entries is not a table, it is a wish.

CHECK YOURSELF. Suppose 2×2 square tiles are also allowed. Find the new recurrence, the first five values, and a closed form.

Answer. The leftmost column is now covered by a vertical domino (leaving $N-1$), two horizontal dominoes (leaving $N-2$), or one 2×2 square (also leaving $N-2$). So $D_N = D_{N-1} + 2D_{N-2}$ with $D_0 = D_1 = 1$, giving 1,1,3,5,11. The characteristic equation $r^2 - r - 2 = 0$ has roots 2 and -1, and fitting the two starting values gives $D_N = \frac{2^{N+1}+(-1)^N}{3}$. Check $N = 4$: $\frac{32+1}{3} = 11$ ✓.

4.9 From Brute Force to Better: a Timed Comparison

Chapter 1 promised that efficiency is not a matter of taste, and Chapter 2's growth table showed the numbers on paper. This last worked problem shows them on a real machine.

4.9.1 Worked Problem L — does any pair sum to x?

Problem. Given n integers $a_0, \dots, a_{n-1}$ and a target x, decide whether there are two different positions $i \neq j$ with $a_i + a_j = x$. Write a brute-force algorithm and a better one, prove both correct, analyze both, and time them on inputs of size 2,000 to 64,000.

What it tests. Pricing a brute-force algorithm (Section 1.4.1), improving it with sorting (Chapter 3's Merge Sort, or the library sort) and binary search (Section 2.14), and reading measured running times against the predictions of asymptotic analysis.

Solution. *The brute force.* Try every pair $i < j$. There are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs, so the worst case — when no pair works and every pair must be tried — costs $\Theta(n^2)$. Correctness is immediate: it examines every pair.

The better algorithm. Sort the array into increasing order. Then, for each position i, the partner we need is $x - a_i$, and since the array is sorted we can look for it by binary search in the part to the right of i. **Pseudocode 4.9** is the procedure; the search is the variant that finds the *first* position whose value is at least the one wanted.

Pseudocode 4.9 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
```

Correctness. Sorting only reorders the values, so the question has the same answer before and after. If the procedure returns TRUE at some i, then $a[lo] = want = x - a[i]$ with $lo > i$: two different positions summing to x. Conversely, suppose some pair exists; after sorting its two values sit at positions $p < q$ with $a[p] + a[q] = x$. When the loop reaches $i = p$, the value $want = a[q]$ occurs in the searched range $a[p+1..n-1]$. By the invariant on line 5, when the search ends lo is the first position in that range whose value is at least $want$; since $a[q] = want$ is in the range, that first position holds exactly $want$, and line 9 returns TRUE. So the procedure returns TRUE exactly when a pair exists.

Cost. Line 1 costs $\Theta(n\lg n)$. Each of the n binary searches costs $O(\lg n)$. Total $\Theta(n\lg n)$.

Hand check. Take $a = \langle 8,3,6,3,9,1 \rangle$, the array of Worked Problem B. Sorted: $\langle 1,3,3,6,8,9 \rangle$. For $x = 12$: $i = 0$ wants 11, absent; $i = 1$ wants $12 - 3 = 9$, which the search finds at position 5. TRUE $(3 + 9)$. For $x = 16$: the wanted partners for $i = 0, \dots, 5$ are 15,13,13,10,8,7. None of them is found to the right of its i. The only near miss is $i = 4$, which wants $16 - 8 = 8$; but the search range is $a[5..5] = \langle 9 \rangle$, and the 8 at position 4 may not be paired with itself. FALSE.

Program. **Listing 4.17** implements both algorithms and times them with `std::chrono::steady_clock`. The data are n distinct odd numbers in a shuffled order, and the target is $x = 1$. The sum of two odd numbers is even, so no pair can ever reach 1: both algorithms are forced to do their full worst-case work, which is exactly what we want to measure. The random generator has a fixed seed, so every run uses the same data.

Listing 4.17 Brute force against sort-and-search for the pair-sum problem, timed on inputs of doubling size.

```
#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 4.17 (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;
}
```

Compiled with `g++ -std=c++17 -O2 -Wall` and run once, it printed

```
     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
```

Your times will differ — they depend on the processor, the compiler and whatever else the machine is doing — but their *ratios* will not differ much, and the ratios are what the analysis predicts. **Table 4.12** computes them from this run.

Table 4.12 Measured times from **Listing 4.17**. 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

Reading the table. For a $\Theta(n^2)$ algorithm, doubling n should multiply the time by $2^2 = 4$; the measured ratios are 3.87 to 4.23. For a $\Theta(n\lg n)$ algorithm, doubling n multiplies the time by $2 \cdot \frac{\lg 2n}{\lg n}$, which is slightly more than 2 (for $n = 32{,}000$ it is $2 \times \frac{15.97}{14.97} \approx 2.13$); the measured ratios scatter around 2, as small timings always do. The last column is the lesson: the better algorithm's

advantage is not a fixed percentage, it *grows with* n — five times faster at 2,000 elements, a hundred and forty-five times faster at 64,000.

We can also extrapolate. At $n = 10^6$ the brute force should take about $670.66 \times \left(\frac{10^6}{64{,}000}\right)^2 \approx 670.66 \times 244 \approx 164{,}000$ ms — nearly three minutes — while the better algorithm should take about $4.631 \times \frac{10^6}{64{,}000} \times \frac{\lg 10^6}{\lg 64{,}000} \approx 4.631 \times 15.6 \times 1.25 \approx 90$ ms. Section 4.2.3's budget said the same thing without a stopwatch: $\frac{n^2}{2} = 5 \times 10^{11}$ pair checks is far over 10^8, and $n\lg n \approx 2 \times 10^7$ is comfortably under.

Why is the measured advantage at $n = 64{,}000$ "only" 145, when the operation counts differ by a factor of about $\frac{n^2/2}{2n\lg n} = \frac{n}{4\lg n} \approx 1{,}000$? Because the two algorithms' basic operations cost different amounts. The brute force's inner loop is two array reads, an addition and a comparison over consecutive memory — the fastest thing a processor does, and one the compiler can vectorize. A sort and a binary search jump around in memory and branch unpredictably. Asymptotic notation hides exactly these constants, which is why Chapter 3 insisted they be dropped from *comparisons of growth* and never from *predictions of seconds*.

Answer. Brute force: correct, $\Theta(n^2)$. Sort and search: correct, $\Theta(n\lg n)$. Measured: about $4\times$ per doubling against about $2\times$ per doubling, and a speed-up that grows from 5 to 145 over the measured range.

> **CHECK YOURSELF.** After sorting, the search loop can be replaced by two pointers: $i = 0$, $j = n - 1$; while $i < j$, if $a_i + a_j = x$ answer TRUE, if the sum is too small increase i, otherwise decrease j. What does this cost, and why is it correct?
>
> *Answer.* Each step moves one pointer inward, so there are at most $n - 1$ steps: $\Theta(n)$ after the $\Theta(n\lg n)$ sort. It is correct because of an invariant: *no pair using a position outside* $[i, j]$ *sums to* x. If $a_i + a_j < x$, then a_i plus any element up to a_j is also too small, so position i can be discarded; symmetrically for j when the sum is too large.

4.10 The Problems of This Chapter Compared

Having worked fourteen problems, step back and ask what each one used. **Table 4.13** lays them side by side.

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)	$\Theta(n^2)$
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)	$\Theta(n \lg n)$

What is common to all of them.

1. **Every one has an obvious method that is either wrong or too slow.** Building the LDCSUMAB list, looping over k in TOHU, evaluating each power separately in Horner's problem, trusting a pattern for a_1, guessing $U(n) \leq cn$, applying the Master Theorem in its gap, trying every pair. Much of the value of each solution lies in *saying why* the obvious method fails.
2. **Every one yields to the table of small cases.** Even the proofs started from tables: the n column of **Table 4.2**, the a_k column of **Table 4.3**, the level totals of **Table 4.9**.
3. **Every answer was checked twice.** A proof and a program; a formula and a hand trace; one method of solving a recurrence and another.

> **THE PATTERN BEHIND THE PATTERN.** Almost every problem in this chapter is solved by the same two moves, in the same order. **Move 1: compute small cases by hand and tabulate them. Move 2: reorganize the computation so that the work per answer drops** — by telescoping a sum, by inverting a counting formula, by carrying a running quantity through a loop, by halving the problem at every step, or by sorting so that a search replaces a scan. Then prove that the reorganized computation still gives the right answer. If you practice only those moves, you will recognize most of the problems you meet in the rest of this book.

4.11 Profitina in Practice

Profitina is an AI-visibility platform — GEO, *Generative Engine Optimization* — live in production at `app.profitina.com`. It scans a brand's website, scores how ready that site is to be *cited by an AI answer engine* from real signals such as content extractability, structured data and the technical credibility of the site, and then helps fix what it finds. Its historical-analytics dashboard keeps a weekly score trend per brand, and its scheduler re-scans each site automatically at a cadence that depends on the customer's subscription tier. The ideas of this chapter appear in it in plain sight.

The complexity budget is a design rule, not a contest trick. A scan is a pipeline of measurements, and every stage has a budget in exactly the sense of **Table 4.1**: the number of pages on a site, the number of signals per page and the number of sites due for a re-scan in the next hour multiply together, and the product must fit in the time available. The first question asked of any new signal is the question of Step 3: *what does it cost per page, and how many pages are there?*

Projections come from closed forms. The dashboard shows, for each brand, how many automatic re-scans its tier will run over a chosen horizon. Simulating the calendar day by day costs $\Theta(\text{days})$ per brand and must be repeated for every brand on every dashboard load; a closed form costs $\Theta(1)$. That is the move of TOHU and Laundry Tokens — spend the algebra once so that the machine never has to loop.

The weekly trend is a growing table, and the best stretch of it is a one-pass scan. A brand owner wants to know *when* the visibility work paid off: "your score rose from 41 to 68 over the quarter" is less useful than "the biggest sustained improvement ran from week 6 to week 13". Finding that stretch over the week-on-week changes is the maximum-sum-window problem, which the case study below solves with prefix sums and a loop invariant. The table gains a row every week and never shrinks, so the difference between a $\Theta(w^2)$ search over all windows and a $\Theta(w)$ scan grows exactly as Worked Problem L's timings did.

Near-duplicate pages are a pair problem. Sites often serve almost the same content at several addresses, and a fair score must not count it twice. Comparing every page with every other page is the brute force of Worked Problem L, $\Theta(n^2)$ comparisons for n pages; reducing each page to a canonical key first and then sorting or hashing the keys brings the work down to about $\Theta(n\lg n)$ — the same reorganization, applied to documents instead of integers.

Checking twice is a release gate. Every solution in this chapter was confirmed a second way: a brute-force checker for LDCSUMAB, exact fractions for TOHU, a program for every recurrence. Profitina applies the same discipline at scale: its full automated test suite, well over a thousand tests, must pass before any release, and a release reports success only after the deployed version answers its health check.

And one rule outranks the algorithms. The platform *advises*; the customer *decides*. A score is an estimate from measured signals, never a guarantee of being cited, and the product presents it that way.

4.11.1 The worked case study: Profitina Laundry

Profitina Laundry is the teaching case study of this book — a small laundry in Keputih, Surabaya, not the real product. The owner has a fortnight of daily net profit in rupiah and wants to know which stretch of consecutive days to repeat as a promotion. The rule: the most profitable

contiguous stretch; if two stretches tie, prefer the longer one, because a longer promotion is easier to advertise; if they still tie, prefer the earlier.

The brute force. A stretch is a pair of days $i \leq j$, and with $n = 14$ days there are $\frac{14 \cdot 15}{2} = 105$ stretches. Summing each from scratch costs $\Theta(n^3)$ in total; keeping a running sum as j grows costs $\Theta(n^2)$. For a fortnight that is instant. For a year of daily data across a chain of branches, it is not.

The better algorithm, from Chapter 3's summations. Define the prefix sums $P[0] = 0$ and $P[j] = p_1 + \cdots + p_j$, the running total after day j. The profit of the stretch $i..j$ is then a single subtraction, $P[j] - P[i-1]$, just as a telescoping sum collapses to its two ends. For a fixed last day j, the best stretch ending there starts right after the day at which the running total was *smallest* before j. So one left-to-right pass suffices, carrying one extra variable with a loop invariant: *at the start of the iteration for day j,* `argmin` *is the earliest index in* $0..j-1$ *at which P is smallest.* Updating `argmin` only on a strict `<` is what keeps it the *earliest*, which enforces the "prefer the longer" rule; comparing lengths on a tie enforces it for the reported window. This is the classic maximum-subarray scan popularized by Bentley [6] and analyzed in Cormen et al. [1]. **Listing 4.18** implements it and runs a deliberately tied dataset as well, so that the tie-breaking rule can be seen at work.

Listing 4.18 Profitina Laundry — the most profitable contiguous stretch of days, by the prefix-minimum method, with the tie-breaking rule demonstrated on a second dataset.

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

struct Window { long long total; int first, last; };

// Maximum-sum window over p[0..n-1]; ties broken by the LONGEST window, then the
// EARLIEST start. Reported in 1-based day numbers. Runs in Theta(n) time.
Window bestWindow(const vector<long long>& p) {
    int n = (int)p.size();
    vector<long long> P(n + 1, 0);
    for (int i = 0; i < n; ++i) P[i + 1] = P[i] + p[i];   // P[j] = profit of days 1..j
    Window w{0, 0, -1};                      // the empty window: total 0, no days at all
    int argmin = 0;                          // earliest i < j at which P[i] is smallest
    for (int j = 1; j <= n; ++j) {
        long long val = P[j] - P[argmin];   // best window that ends at day j
        int len  = j - argmin;
        int wlen = (w.last < 0) ? 0 : (w.last - w.first + 1);
        if (val > w.total || (val == w.total && len > wlen)) {
            w.total = val; w.first = argmin + 1; w.last = j;
        }
        if (P[j] < P[argmin]) argmin = j;   // strict '<' keeps the EARLIEST minimum
    }
    return w;
}

string rupiah(long long v) {
    bool neg = v < 0; if (neg) v = -v;
    string s = to_string(v), out;
    int c = 0;
    for (int i = (int)s.size() - 1; i >= 0; --i) {
        out += s[i];
        if (++c % 3 == 0 && i > 0) out += '.';
    }
    if (neg) out += '-';
    return "Rp " + string(out.rbegin(), out.rend());
}
```

Listing 4.18 (Continued) *Profitina Laundry — the most profitable contiguous stretch of days, by the prefix-minimum method, with the tie-breaking rule demonstrated on a second dataset.*

```
void report(const string& title, const vector<string>& day, const vector<long long>& p) {
    cout << title << "\n" << string(58, '-') << "\n";
    long long run = 0;
    for (size_t i = 0; i < p.size(); ++i) {
        run += p[i];
        cout << left << setw(9) << day[i] << right << setw(15) << rupiah(p[i])
             << "   running " << setw(15) << rupiah(run) << "\n";
    }
    cout << string(58, '-') << "\n";
    Window w = bestWindow(p);
    if (w.last < 0) { cout << "No stretch of days is profitable.\n\n"; return; }
    cout << "Best stretch : day " << w.first << " to day " << w.last
         << "  (" << day[w.first - 1] << " .. " << day[w.last - 1] << ")\n";
    cout << "Net over it  : " << rupiah(w.total) << "\n";
    cout << "Length       : " << (w.last - w.first + 1) << " days\n\n";
}

int main() {
    vector<string> day = {"Mon 1","Tue 2","Wed 3","Thu 4","Fri 5","Sat 6","Sun 7",
                          "Mon 8","Tue 9","Wed 10","Thu 11","Fri 12","Sat 13","Sun 14"};
    vector<long long> profit = { -90000, 120000, -45000,  80000, -30000,  95000, 210000,
                                 -400000,  55000, -70000,  60000,  40000, 150000,-200000 };
    report("Profitina Laundry - Keputih branch, 14-day net profit", day, profit);

    // A deliberately tied case, to show the tie-breaking rule at work.
    vector<string> d3 = {"Mon 1","Tue 2","Wed 3"};
    vector<long long> t3 = { 30000, -30000, 30000 };
    report("Tie-break demonstration: three windows all worth Rp 30.000", d3, t3);
    return 0;
}
```

Compiled and run, it prints:

```
Profitina Laundry - Keputih branch, 14-day net profit
-------------------------------------------------------
Mon 1         Rp -90.000   running      Rp -90.000
Tue 2         Rp 120.000   running       Rp 30.000
Wed 3         Rp -45.000   running      Rp -15.000
Thu 4          Rp 80.000   running       Rp 65.000
Fri 5         Rp -30.000   running       Rp 35.000
Sat 6          Rp 95.000   running      Rp 130.000
Sun 7         Rp 210.000   running      Rp 340.000
Mon 8        Rp -400.000   running      Rp -60.000
Tue 9          Rp 55.000   running       Rp -5.000
Wed 10        Rp -70.000   running      Rp -75.000
Thu 11         Rp 60.000   running      Rp -15.000
Fri 12         Rp 40.000   running       Rp 25.000
Sat 13        Rp 150.000   running      Rp 175.000
Sun 14       Rp -200.000   running      Rp -25.000
-------------------------------------------------------
Best stretch : day 2 to day 7  (Tue 2 .. Sun 7)
Net over it  : Rp 430.000
Length       : 6 days

Tie-break demonstration: three windows all worth Rp 30.000
-------------------------------------------------------
Mon 1          Rp 30.000   running       Rp 30.000
Tue 2         Rp -30.000   running            Rp 0
Wed 3          Rp 30.000   running       Rp 30.000
-------------------------------------------------------
Best stretch : day 1 to day 3  (Mon 1 .. Wed 3)
Net over it  : Rp 30.000
Length       : 3 days
```

The first answer, days 2 to 7 worth Rp 430.000, is easy to confirm from the running column, which is exactly the prefix sum $P[j]$: the lowest running total before the peak is $-90{,}000$ at day 1, the peak is 340,000 at day 7, and $340{,}000 - (-90{,}000) = 430{,}000$. The Monday-8 collapse of Rp 400.000 — a broken dryer — cuts the fortnight in two, which is exactly why the best stretch stops at day 7 rather than running on. The second answer shows the rule at work: three different windows are worth Rp 30.000 each (day 1 alone, day 3 alone, and days 1 to 3), and the program returns the longest. The whole computation is $\Theta(n)$ time and $\Theta(n)$ space for the prefix array, against $\Theta(n^2)$ for the running-sum brute force. As always, the program advises; the owner decides which promotion to run.

4.12 Common Mistakes, and How to Avoid Them

1. Writing the code first. Most wrong programs are wrong before the first line is typed, because the formula or invariant behind them was never written down. *Fix:* Section 4.2, Steps 1 to 6, before a single line.

2. Skipping the small cases. The table of the first five values solved LDCSUMAB, TOHU, Laundry Tokens and the $2 \times N$ strip, and exposed the exceptional a_1 in TOHU. *Fix:* build the table, always, even when you think you already see the answer.

3. Trusting a pattern without proving it. Five rows of a table are evidence, not a proof. TOHU's $a_k = \frac{1}{k(k+1)(k+2)}$ holds from $k = 2$ but not at $k = 1$. *Fix:* prove the pattern — by differencing, telescoping, induction or an invariant — and let the proof tell you where it holds.

4. Ignoring the complexity budget. A correct $\Theta(n^2)$ solution at $n = 10^5$ is too slow, and a correct $\Theta(k)$ list at $k = 10^9$ does not fit in memory. *Fix:* read the constraint, consult **Table 4.1**, and write the budget calculation down.

5. Off-by-one at a boundary. Ceiling versus floor in LDCSUMAB; `right = mid` versus `right = mid - 1` in binary search; a loop from i to j running $j - i + 1$ times, not $j - i$; F_N versus F_{N+1} for the strip. *Fix:* test the boundary cases explicitly — every triangular number and its successor, a window of one element, $N = 0$ and $N = 2$.

6. An invariant without all three parts. Proving initialization and maintenance but forgetting termination proves nothing about the output. *Fix:* write the three headings — *Initialization, Maintenance, Termination* — and fill in each one, as in Worked Problem A.

7. Letting the constant drift in a substitution proof. Deriving $U(n) \leq cn + 1$ from the hypothesis $U(n) \leq cn$ and concluding "so $U(n) = O(n)$" is invalid. *Fix:* the induction must reproduce the hypothesis exactly; if it does not, strengthen the hypothesis by subtracting a lower-order term (Worked Problem G).

8. Constants that depend on n. In an asymptotic proof, c and n_0 must be fixed numbers. "Take $c = n$" proves nothing. *Fix:* name the constants explicitly, as in Worked Problem E, and check them numerically.

9. Applying the Master Theorem where it does not apply. A logarithmic factor is not a polynomial one: $T(n) = T(n/2) + \lg n$ is in the gap, and $T(n) = 2T(n-1) + 1$ is not of the master form at all. *Fix:* compute ε explicitly; if no positive ε exists, unroll the recurrence or draw the tree.

10. Integer arithmetic where real arithmetic was meant. `2 * n / (n + 1)` is 1 for every $n \geq 1$; `3/4` is 0. *Fix:* convert to `double` before dividing, and write real constants with a decimal point.

11. Overflowing silently. $2(n+1)(n+2)$ overflows an `int` from $n = 32{,}767$; $n(n+1)$ is within a whisker of the `int` limit in LDCSUMAB; a product of three residues modulo $10^9 + 7$ overflows a `long long`. *Fix:* estimate the largest intermediate value before choosing the type, and reduce modulo p after every multiplication.

12. Trusting floating point for integer work. `sqrt` on a perfect square may land a hair either side of the integer when the argument exceeds 2^{53}. *Fix:* use a floating-point root only as a first guess and correct it with integer tests, as in **Listing 4.1**, or avoid it with binary search, as in **Listing 4.2**.

13. Using `float` for eleven decimals. Seven significant digits cannot print eleven. *Fix:* `double` for everything numerical in this chapter.

14. Reading timings without ratios. A single timing says little; "brute force took 670 ms" depends on the machine. *Fix:* time a sequence of doubling sizes and read the *ratios*, as in **Table 4.12**; those are what the analysis predicts.

4.13 Chapter Summary

This chapter consolidated Chapters 1 to 3 by working problems rather than presenting new theory.

The **seven-step method** of Section 4.2 is the transferable part: copy the formats; name the given and the wanted; turn the constraints into a time budget; compute the first cases by hand; name

the family; write the invariant, recurrence or formula, and prove it; only then write the program — checking overflow, precision and input speed.

The two **judge problems** showed the method at full length. **LDCSUMAB** turned a position in an infinite sorted list into a pair of exponents: two claims fixed the order of the list, the arithmetic series counted its blocks, the quadratic formula and a ceiling inverted the count, a loop invariant made the floating-point guess safe, and binary search gave a second solution with no floating point at all. **TOHU** recovered a hidden sequence by differencing a weighted sum, found that the first term is an exception and why, summed the rest by telescoping to $S_n = \frac{3}{4} - \frac{1}{2(n+1)(n+2)}$, and confirmed the result by induction and by exact rational arithmetic.

The twelve **worked problems** practiced one tool each: a loop invariant for Horner's rule; exact traces of Insertion Sort (where shifts equal inversions) and binary search (where $\lfloor \lg n \rfloor + 1$ probes suffice); the exact count $\frac{n(n+1)(n+2)}{6}$ of a triple loop; asymptotic claims proved and disproved with explicit c and n_0; thirteen functions ranked by growth, with the warning that small n can mislead; a substitution proof that needed a strengthened hypothesis; a recursion tree summed exactly to $\frac{16n^2 - 3^{k+1}}{13}$; the Master Theorem in all three cases and in its gap; a telescoping closed form; the Fibonacci numbers hidden in a tiling problem and computed by fast doubling; and a timed race in which a $\Theta(n \lg n)$ algorithm pulled away from a $\Theta(n^2)$ one by a factor that doubled with every doubling of n.

Two moves did most of the work: **tabulate the small cases**, then **reorganize the computation** so that the cost per answer drops — and prove that the reorganized computation is still right.

Looking ahead. Chapter 5 returns to the main line of the book with Heapsort and Quicksort — two sorting algorithms that both run in $\Theta(n \lg n)$ on typical input for completely different reasons, and whose analysis uses the loop invariants, summations and recurrences practiced here.

4.14 Exercises

4.1 Prove, with a loop invariant, that the following procedure returns the largest element of $A[1..n], n \geq 1$: `m = A[1]; for i = 2 to n: if A[i] > m then m = A[i]; return m`.

4.2 Find the exact number of times the body executes, and its Θ class: (a) `for i = 1 to n: for j = 1 to 2i: body`; (b) `for i = 1 to n: { j = 1; while j < i: { j = 2j; body } }`. For (b), also give the exact count for $n = 8$.

4.3 Trace Insertion Sort on the reverse-sorted array $\langle 6,5,4,3,2,1 \rangle$. Give the array after each outer iteration, the total number of shifts, and $\sum_j t_j$. Relate the shifts to the number of inversions.

4.4 In the array $A[i] = i$ for $i = 1, \ldots, 1000$, trace binary search for $q = 1000$ by listing the successive values of mid. What is the largest number of probes any search in this array can need?

4.5 Prove $3n^3 - 100n^2 = \Theta(n^3)$ with explicit constants c_1, c_2, n_0.

4.6 Is $2^n = \Theta(2^{n/2})$? Is $2^{n/2} = O(2^n)$? Prove your answers.

4.7 Solve $T(n) = 2T(n/2) + n^2$, $T(1) = 1$, for n a power of 2, by a recursion tree. Give the exact value and the Θ class, and confirm with the Master Theorem.

4.8 Apply the Master Theorem, or explain why it does not apply and solve otherwise: (a) $T(n) = 2T(n/4) + 1$; (b) $T(n) = 2T(n/4) + n$; (c) $T(n) = 4T(n/2) + n^2 \lg n$; (d) $T(n) = 6T(n/3) + n^2 \lg n$.

4.9 LDCSUMAB asks for *unranking*. Write the inverse, *ranking*: given $0 \le m < n$, compute the position k of $2^m + 2^n$ in the sorted list. Check it on (1,3) and on (38439,44721).

4.10 Change TOHU's condition to $a_1 + 2a_2 + \cdots + ka_k = \frac{k}{k+2}$. Find a_k and S_n in closed form, and explain why this time the first term is not an exception.

4.11 Using **Table 4.12**, predict the running times of both algorithms of Worked Problem L at $n = 128{,}000$. Then explain why, for very small n, the brute force can be the faster of the two.

4.12 Compose a problem in the style of Section 4.8 whose answer is a closed form found by telescoping, and solve it with the seven steps of Section 4.2.

4.15 Solutions to the Exercises

4.1 *Invariant:* at the start of the iteration with index i, $m = \max(A[1..i-1])$. *Initialization:* $i = 2$ and $m = A[1] = \max(A[1..1])$. ✓ *Maintenance:* if $A[i] > m$ the body sets $m = A[i]$, otherwise leaves m; either way m becomes $\max(m, A[i]) = \max(A[1..i])$, which is the invariant for the next index $i + 1$. ✓ *Termination:* the loop ends with $i = n + 1$, where the invariant says $m = \max(A[1..n])$. ✓ For $n = 1$ the loop does not run and $m = A[1]$ is correct.

4.2 (a) For each i the inner loop runs $2i$ times, so the total is $\sum_{i=1}^{n} 2\,i = 2 \cdot \frac{n(n+1)}{2} = n(n+1) = \Theta(n^2)$. (b) For a given i, j takes the values 1,2,4, … and the body runs once per doubling until $j \ge i$, which is $\lceil \lg i \rceil$ times (zero times for $i = 1$). The total is $\sum_{i=1}^{n} \lceil \lg i \rceil$. For $n = 8$: $0 + 1 + 2 + 2 + 3 + 3 + 3 + 3 = 17$. Since $\lg i \le \lceil \lg i \rceil < \lg i + 1$, the total lies between $\lg(n!)$ and $\lg(n!) + n$, so by Worked Problem E(d) it is $\Theta(n \lg n)$.

4.3 After $j = 2,3,4,5,6$ the array is $\langle 5,6,4,3,2,1 \rangle$, $\langle 4,5,6,3,2,1 \rangle$, $\langle 3,4,5,6,2,1 \rangle$, $\langle 2,3,4,5,6,1 \rangle$, $\langle 1,2,3,4,5,6 \rangle$. Iteration j shifts all $j - 1$ earlier elements, so the shifts are $1 + 2 + 3 + 4 + 5 = 15 = \frac{6 \cdot 5}{2}$, and $t_j = j$, giving $\sum_{j=2}^{6} t_j = 2 + 3 + 4 + 5 + 6 = 20$. Every one of the $\binom{6}{2} = 15$ pairs is an inversion in a reversed array, so shifts equal inversions, as Worked Problem B proved in general. This is the worst case of Section 2.11.2: $\Theta(n^2)$.

4.4 The midpoints are 500,750,875,938,969,985,993,997,999,1000: ten probes, the last one successful. The largest number of probes any search can need is $\lfloor \lg 1000 \rfloor + 1 = 9 + 1 = 10$, since $2^9 = 512 \le 1000 < 1024 = 2^{10}$; so $q = 1000$ is a worst case.

4.5 Upper: $3n^3 - 100n^2 \le 3n^3$ for all $n \ge 1$, so $c_2 = 3$. Lower: $2n^3 \le 3n^3 - 100n^2 \Leftrightarrow 100n^2 \le n^3 \Leftrightarrow n \ge 100$, so $c_1 = 2$ with $n_0 = 100$. Together: $c_1 = 2$, $c_2 = 3$, $n_0 = 100$. (For $n < 100$ the function is not even positive at, say, $n = 10$, where it equals $3000 - 10000 = -7000$; the threshold is essential.)

4.6 $2^n = \Theta(2^{n/2})$ is false: if $2^n \le c\,2^{n/2}$ for all $n \ge n_0$, then dividing by $2^{n/2}$ gives $2^{n/2} \le c$ for all $n \ge n_0$, which fails once $n > 2\lg c$. On the other hand $2^{n/2} \le 1 \cdot 2^n$ for all $n \ge 0$, so $2^{n/2} = O(2^n)$ with $c = 1$, $n_0 = 0$. As in Worked Problem E(c), a constant factor in the exponent changes the base: $2^{n/2} = \left(\sqrt{2}\right)^n$.

4.7 Level i has 2^i nodes of size $n/2^i$, each costing $(n/2^i)^2$, so the level costs $2^i \cdot n^2/4^i = n^2/2^i$. With $n = 2^k$, the internal levels $i = 0, \ldots, k-1$ sum to $n^2\left(2 - \frac{2}{2^k}\right) = 2n^2 - 2n$, and the $2^k = n$ leaves add n. So $T(n) = 2n^2 - n$ exactly. Check: $T(2) = 2 \cdot 1 + 4 = 6 = 8 - 2$ ✓; $T(4) = 2 \cdot 6 +$

$16 = 28 = 32 - 4$ ✓. $\Theta(n^2)$. Master Theorem: $n^{\log_2 2} = n$, $f(n) = n^2 = \Omega(n^{1+1})$, regularity $2(n/2)^2 = \frac{1}{2}n^2$ with $c = \frac{1}{2}$: Case 3, $\Theta(n^2)$ ✓.

4.8 (a) $\log_4 2 = \frac{1}{2}$; $f(n) = 1 = O(n^{1/2-1/2})$: Case 1, $\Theta(\sqrt{n})$. (b) $f(n) = n = \Omega(n^{1/2+1/2})$; regularity $2 \cdot \frac{n}{4} = \frac{1}{2}n$, $c = \frac{1}{2}$: Case 3, $\Theta(n)$. (c) $\log_2 4 = 2$ and $f(n) = n^2 \lg n$ exceeds n^2 only by a logarithmic factor: the gap, and the theorem does not apply. By the tree, level i costs $4^i(n/2^i)^2 \lg(n/2^i) = n^2(\lg n - i)$; summing for $i = 0, \dots, \lg n - 1$ gives $n^2 \cdot \frac{\lg n(\lg n+1)}{2}$, plus n^2 leaves: $\Theta(n^2 \lg^2 n)$. (d) $\log_3 6 \approx 1.631$ and $f(n) = n^2 \lg n = \Omega(n^{1.631+0.3})$; regularity: $6f(n/3) = 6 \cdot \frac{n^2}{9} \lg \frac{n}{3} \le \frac{2}{3} n^2 \lg n$, so $c = \frac{2}{3}$: Case 3, $\Theta(n^2 \lg n)$.

4.9 Blocks $1, \dots, n-1$ fill the first $T_{n-1} = \frac{(n-1)n}{2}$ positions, and inside block n the pair with smaller exponent m is the $(m+1)$-th. So $k = \frac{n(n-1)}{2} + m + 1$. Check (1,3): $3 + 1 + 1 = 5$ ✓. Check (38439,44721): $\frac{44721 \cdot 44720}{2} + 38440 = 999{,}961{,}560 + 38{,}440 = 1{,}000{,}000{,}000$ ✓ — the query of Section 4.3.4, inverted.

4.10 Differencing as in Section 4.4.4: $ka_k = \frac{k}{k+2} - \frac{k-1}{k+1} = \frac{k(k+1)-(k-1)(k+2)}{(k+1)(k+2)} = \frac{(k^2+k)-(k^2+k-2)}{(k+1)(k+2)} = \frac{2}{(k+1)(k+2)}$, so $a_k = \frac{2}{k(k+1)(k+2)}$. At $k = 1$ the condition gives $a_1 = \frac{1}{3}$, and the formula gives $\frac{2}{6} = \frac{1}{3}$ too: no exception, because the formula for the previous weighted sum, $\frac{k-1}{k+1}$, gives 0 at $k = 1$, which is exactly the empty sum. By Equation (4.6), $a_k = \frac{1}{k(k+1)} - \frac{1}{(k+1)(k+2)}$, so $S_n = \frac{1}{1 \cdot 2} - \frac{1}{(n+1)(n+2)} = \frac{1}{2} - \frac{1}{(n+1)(n+2)}$. Check $n = 2$: $a_1 + a_2 = \frac{1}{3} + \frac{1}{12} = \frac{5}{12}$, and $\frac{1}{2} - \frac{1}{12} = \frac{5}{12}$ ✓.

4.11 Doubling n from 64,000 multiplies the brute-force time by about 4 and the sort-and-search time by a little over 2: roughly $4 \times 670.66 \approx 2{,}700$ ms against $2.1 \times 4.631 \approx 9.7$ ms, a ratio of nearly 280. For very small n the constants dominate: with $n = 10$ the brute force performs 45 additions and comparisons in a tight loop, while the better algorithm must copy the array, sort it and run ten binary searches, each with its own overhead. The asymptotic advantage only appears once n is large enough for n^2 to swamp those constants — which is why **Table 4.12** shows only a factor of 5 at $n = 2{,}000$.

4.12 One acceptable answer. *Problem SQPYRAMID.* A pyramid of laundry baskets has 1 basket on the top layer, 4 on the second, 9 on the third, and k^2 on the k-th. Given $n \le 10^6$, up to 10^5 times, how many baskets are in the first n layers? *Step 1:* one integer per line in, one integer per line out. *Step 2:* given n; wanted $P_n = \sum_{k=1}^{n} k^2$. *Step 3:* 10^5 queries times 10^6 terms is 10^{11}, over budget; find a closed form. *Step 4:* $P_1, \dots, P_5 = 1,5,14,30,55$. *Step 5:* closed form from a sum. *Step 6:* telescope $(k+1)^3 - k^3 = 3k^2 + 3k + 1$, exactly as in Worked Problem D, to get $P_n = \frac{n(n+1)(2n+1)}{6}$; check $n = 3$: $\frac{3 \cdot 4 \cdot 7}{6} = 14$ ✓. *Step 7:* $O(1)$ per query; the largest answer, at $n = 10^6$, is about 3.3×10^{17}, which fits in a `long long`, but the product $n(n+1)(2n+1) \approx 2 \times 10^{18}$ must be formed before dividing by 6, and it also fits (below 9.22×10^{18}) — so compute in `long long` and divide last, and note that the constraint could not grow much further without overflow.

References

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

[2] K. H. Rosen, *Discrete Mathematics and Its Applications*, 7th ed. New York, NY, USA: McGraw-Hill, 2012.

[3] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4*, 4th ed. Singapore: Lulu, 2020.

[4] Sphere Research Labs, "SPOJ — Sphere Online Judge." [Online]. Available: https://www.spoj.com/

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

[6] J. Bentley, *Programming Pearls*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2000.

The loop invariants, the analysis of Insertion Sort and binary search, the asymptotic notation, the recurrence methods and the Master Theorem used throughout this chapter follow Cormen et al. [1, chs. 2–4]; Horner's rule appears there as a problem at the end of Chapter 2. The summations, the triangular numbers of LDCSUMAB and the differencing and induction arguments of TOHU are standard discrete mathematics [2]. The complexity budget of **Table 4.1** follows the conventions of competitive-programming practice as set out in [3], and both judge problems are hosted on SPOJ [4]. Measuring algorithms against each other with a stopwatch, and the maximum-sum-window scan of the Profitina Laundry case study, are in the spirit of [6]. Telescoping sums, the domino tilings of the $2 \times N$ strip and the fast-doubling identities for Fibonacci numbers are treated at length in [5, chs. 2, 6 and 7].

Chapter 5
Sorting in Practice: Heapsort and Quicksort

By the end of this chapter you will be able to build a binary heap from an unordered array in linear time and prove that the linear bound is correct; sort any array in guaranteed $\Theta(n\lg n)$ time using no extra memory at all; implement a priority queue that inserts, peeks, extracts and re-prioritizes in $O(\lg n)$; write Quicksort from memory, partition an array correctly the first time, and prove the four-region loop invariant that makes the partition work; explain exactly why Quicksort is $\Theta(n^2)$ in the worst case and $\Theta(n\lg n)$ on average, and why the average is what you actually get; randomize the pivot and say precisely what that buys you; and choose, for any concrete sorting problem put in front of you, which of the four algorithms you now know is the right one — and defend the choice with numbers rather than opinion. You will also see why no comparison sort can ever beat $\Omega(n\lg n)$, which tells you that the chapter's two algorithms are, in a real sense, the end of the road.

5.1 Where We Left Off

Chapters 2 and 3 gave us four sorting algorithms and the mathematics to weigh them. Chapter 2 introduced the RAM model and Insertion Sort, and proved Insertion Sort correct with a loop invariant. Chapter 3 gave us O, Ω and Θ; gave us recurrences and three ways to solve them — substitution, the recursion tree, and the Master Theorem; and ended by deriving Merge Sort from nothing and proving that $T(n) = 2T(n/2) + \Theta(n)$ has the solution $\Theta(n\lg n)$.

Before we add anything new, let us lay the four algorithms side by side and look honestly at what we have.

5.1.1 The toolkit so far

Table 5.1 is the inventory, and it is worth reading slowly.

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

Two of those columns need unpacking before we go on, because this chapter turns on them.

In-place means the algorithm rearranges the elements inside the array it was given, using only a fixed number of extra variables — a loop counter, a temporary for a swap, a pivot value — no matter how large n is. Insertion Sort is in-place: it shuffles elements rightwards inside A and drops the key into the gap. Merge Sort is not: `MERGE` cannot combine two sorted halves without somewhere to put the combined result, and that somewhere is an array of n elements.

Extra space puts a number on the same idea. $O(1)$ means "a constant amount, independent of n". $O(n)$ means "an amount that grows in proportion to n" — for an array of ten million 4-byte integers, that is another forty megabytes of memory you must have available, and another forty megabytes of data the processor must push through its memory bus.

5.1.2 The frustrating gap

Now read **Table 5.1** again looking only at two columns: **worst case** and **in-place**.

Every algorithm that sorts in-place is $\Theta(n^2)$. The only algorithm that reaches $\Theta(n\lg n)$ is not in-place. There is a hole in the table exactly where we would most like an entry to be:

> **THE DREAM.** A sorting algorithm that runs in $O(n\lg n)$ time **and** sorts in place, using only $O(1)$ extra memory. Nothing we have met so far does both. This chapter delivers two algorithms that fill the hole: **Heapsort**, which is $\Theta(n\lg n)$ in the worst case and uses $O(1)$ extra space, and **Quicksort**, which is $\Theta(n\lg n)$ on average, uses $O(\lg n)$ extra space, and is the fastest general-purpose sort in practice.

5.1.3 Why $\Theta(n\lg n)$ is the target, and what sorting costs in a real system

It is worth being concrete about why the gap between n^2 and $n\lg n$ is not an academic nicety.

Take $n = 10{,}000{,}000$ — ten million records, which is a small table by the standards of any production database. Then

$$n^2 = 10^{14} = 100{,}000{,}000{,}000{,}000$$

and, since $\lg(10^7) \approx 23.25$,

$$n\lg n \approx 10^7 \times 23.25 = 232{,}500{,}000 \approx 2.3 \times 10^8.$$

The ratio is

$$\frac{10^{14}}{2.3 \times 10^8} \approx 430{,}000.$$

A computer that performs 10^9 elementary operations per second — a fair figure for a single core doing simple array work — would finish the $n\lg n$ algorithm in about a quarter of a second. The n^2 algorithm on the same machine would need $10^{14}/10^9 = 10^5$ seconds, which is **twenty-seven hours**. This is not "a bit slower". It is the difference between a page that loads and a page that never loads.

And why is $n\lg n$ the *target*, rather than, say, n? Because, as we shall prove in Section 5.11 at the end of this chapter, **no algorithm that sorts by comparing pairs of elements can do better than $\Omega(n\lg n)$ in the worst case.** The bound is not a limit of our cleverness; it is a property of the problem. Once an algorithm reaches $\Theta(n\lg n)$ it has reached the floor, and the only remaining competition is over constant factors, memory traffic and guarantees. That competition is precisely what Sections 5.9 to 5.12 are about.

Finally, a word about where sorting actually turns up, because beginners often imagine it is a thing you do once to a list of names. In a real system sorting is everywhere, usually hidden:

- A database executing `ORDER BY` sorts. So does a database building an index, and so does a database performing a *sort-merge join* of two tables.
- A search engine ranking results sorts. A recommendation feed sorts. An analytics dashboard drawing a "top 20" chart sorts.
- Computing a **median**, a **percentile** or a **duplicate count** is most simply done by sorting first.
- Graph algorithms you will meet in Chapters 11 and 13 — Kruskal's minimum spanning tree, Dijkstra's shortest path — either sort the edges outright or use the priority queue of Section 5.7, which is the same data structure.
- Every `GROUP BY` that is not done with a hash table is done by sorting.

A production service that sorts on every request cannot afford $\Theta(n^2)$, and it cannot afford to be surprised by a worst case either. Both halves of that sentence matter, and each of this chapter's two algorithms answers one of them.

5.2 The Selection Sort Insight

The path to Heapsort starts with a piece of self-criticism aimed at Selection Sort.

Recall what Selection Sort does. It works from the right end of the array toward the left. At each step it has already placed the largest elements in the tail $A[i+1 \dots n]$, and it must now find the largest of what remains and put it at position i. So each iteration consists of two steps:

- **Step A — find the largest.** Scan the whole of $A[1 \dots i]$ and remember the biggest value seen. This examines i elements, so it costs $\Theta(i)$, which is $\Theta(n)$ in the early iterations.
- **Step B — place it.** Swap that element into position i. One exchange. $\Theta(1)$.

There are $n-1$ iterations, so the total is

$$\underbrace{\Theta(n)}_{\text{Step A}} \times \underbrace{(n-1)}_{\text{iterations}} = \Theta(n^2).$$

Look at where the cost is. Step B is free. **All** the cost is in Step A: finding the maximum, over and over, from scratch, each time. Each scan throws away everything it learned. On the second pass the algorithm compares the same pairs it compared on the first pass, having forgotten the answers.

So ask the obvious question: *what if finding the maximum did not cost $\Theta(n)$?* If we could find and remove the maximum in $O(\lg n)$ instead, the total would become

$$O(\lg n) \times (n-1) = O(n \lg n),$$

and the dream of Section 5.1.2 would be achieved without changing the *shape* of the algorithm at all.

> **KEY IDEA.** Sometimes the way to speed up an algorithm is not to change the algorithm but to change the **data structure** it keeps its data in. Selection Sort is slow because a plain array is a terrible place to look for a maximum repeatedly. Store the same numbers in a **binary heap** and the maximum is free, while restoring order after removing it costs only $O(\lg n)$. Same algorithm, different container, quadratic becomes $n \lg n$.

That is the whole plan for the first half of this chapter. Build the container (Section 5.3), build the one operation that keeps it honest (Section 5.4), build the container from scratch cheaply (Section 5.5), and then run Selection Sort on top of it (Section 5.6).

5.3 The Binary Heap: a Tree Hidden Inside an Array

5.3.1 Nearly complete binary trees

We begin with a shape.

A **binary tree** is a collection of nodes in which one node is the *root*, and every node has at most two children, distinguished as the *left* child and the *right* child. The *depth* of a node is the number of edges on the path from the root down to it, so the root has depth 0. A *level* is the set of all nodes at the same depth. The *height* of a node is the number of edges on the longest downward path from that node to a leaf; the height of the tree is the height of its root. A node with no children is a **leaf**.

A binary tree is **nearly complete** when it satisfies two conditions:

1. Every level is completely filled, except possibly the last one.
2. The last level is filled from the left, with no gaps.

> **EXAMPLE.** With ten nodes the shape is forced: level 0 holds 1 node, level 1 holds 2, level 2 holds 4, and the remaining $10 - 7 = 3$ nodes sit at the far left of level 3. There is exactly one nearly complete binary tree on ten nodes, and it has height 3.

That rigidity is the point. Because the shape is forced, we never need to store it. A nearly complete binary tree with n nodes is completely described by the single number n — and that is what lets us hide the tree inside an array with no pointers whatsoever.

5.3.2 The index arithmetic, derived

Number the nodes $1,2,3,\ldots,n$ in **level order**: the root is 1; then the two nodes of level 1, left to right, are 2 and 3; then the four nodes of level 2 are 4,5,6,7; then 8,9,10, … and so on. Store the value of node i in array slot $A[i]$. Slot $A[0]$ is left unused so that the arithmetic comes out clean.

Now the three navigation formulas, which we shall *derive* rather than memorize. **Figure 5.1** shows them written out beside the tree and the array they describe.

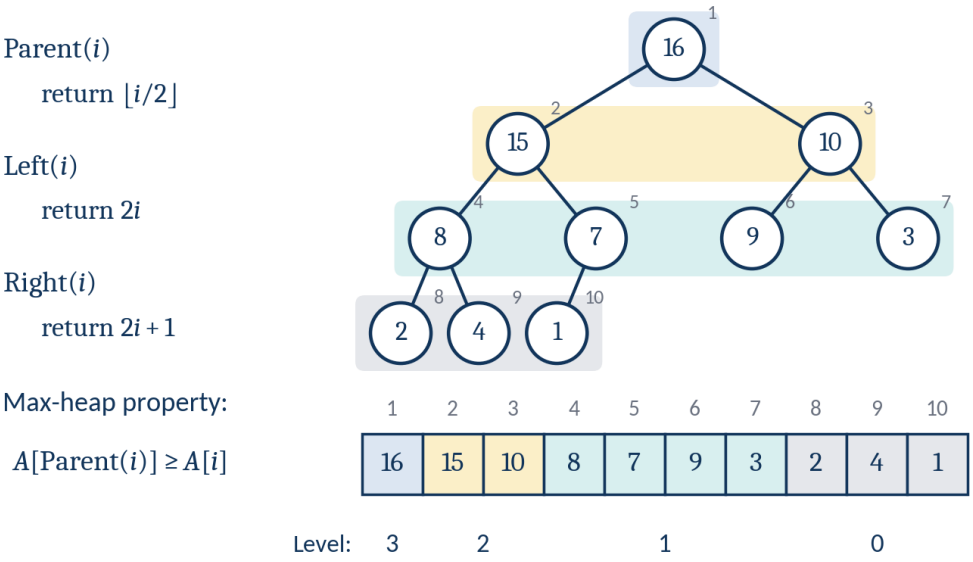


Figure 5.1 The three navigation formulas, the tree they describe, and the array that holds it. The colored bands group the nodes by depth and mark the matching runs of the array beneath; the row labeled "Level" counts from the bottom up, so the root's band is 3 and the bottom row is 0.

Here is where the formulas come from. Level d holds exactly 2^d nodes (level 0 holds $2^0 = 1$, level 1 holds $2^1 = 2$, and each node of one level fathers two of the next). So the number of nodes strictly above level d is

$$1 + 2 + 4 + \cdots + 2^{d-1} = 2^d - 1,$$

by the geometric sum of Chapter 3. Therefore the first node of level d carries the index 2^d, and the nodes of level d run from 2^d to $2^{d+1} - 1$.

Take a node at index i on level d. It is the k-th node of its level, counting from zero, where $i = 2^d + k$. Its two children are on level $d + 1$. The nodes of level $d + 1$ are the children of the nodes of level d in order, two apiece, so the children of the k-th node of level d are the $2k$-th and $(2k + 1)$-th nodes of level $d + 1$. Their indices are therefore

$$2^{d+1} + 2k \quad \text{and} \quad 2^{d+1} + 2k + 1.$$

But $2^{d+1} + 2k = 2(2^d + k) = 2i$, and $2^{d+1} + 2k + 1 = 2i + 1$. That is the derivation:

$$\text{Left}(i) = 2i, \qquad \text{Right}(i) = 2i + 1. \tag{5.1}$$

The parent formula follows by inverting. If j is a child of i then j is either $2i$ or $2i + 1$. In the first case $j/2 = i$ exactly; in the second $j/2 = i + \frac{1}{2}$, whose floor is i. Either way

$$\text{Parent}(j) = \lfloor j/2 \rfloor. \tag{5.2}$$

Together, Equation (5.1) and Equation (5.2) are the whole of the navigation: every move in a heap is one of these three formulas. **Table 5.2** records all three, with a worked instance and a note on why they are so cheap to compute.

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	$\lfloor i/2 \rfloor$	$\text{Parent}(3) = 1$	Right-shift by one bit
Left child of node i	$2i$	$\text{Left}(3) = 6$	Left-shift by one bit
Right child of node i	$2i + 1$	$\text{Right}(3) = 7$	Left-shift, then set the low bit

The "hardware trick" column deserves a sentence. In binary, multiplying by two appends a zero bit, and dividing by two with truncation discards the last bit. A processor does either in one cycle with a shift instruction. Compare this with a tree built out of nodes and pointers, where finding a child means *dereferencing a pointer*: fetching an address from memory, then fetching the node it points at, from wherever in memory that happens to be. A heap has no pointers to chase and no nodes scattered across the address space. It is one flat, contiguous array. That is why heaps are fast in practice as well as in theory.

5.3.3 The heap property

The shape is only half the definition. The other half is an ordering constraint.

> **DEFINITION.** A **binary max-heap** is an array $A[1 \dots n]$, viewed as a nearly complete binary tree by the index arithmetic above, in which every node is at least as large as each of its children:

$$A[\text{Parent}(i)] \geq A[i] \qquad \text{for every } i = 2,3, \dots, n. \tag{5.3}$$

> This inequality is the **max-heap property**. A **min-heap** reverses it: $A[\text{Parent}(i)] \leq A[i]$ for every $i > 1$.

We shall work with max-heaps throughout this chapter, because a max-heap is what Heapsort needs. Min-heaps are what graph algorithms need: Dijkstra's shortest-path algorithm (Chapter 13) and Prim's minimum spanning tree (Chapter 11) both repeatedly ask for the *smallest* remaining key. Everything in this chapter carries over by flipping every $\geq$ to $\leq$ and every "largest" to "smallest".

Note carefully what the heap property of Equation (5.3) does **not** say. It says nothing about the relationship between two siblings, nothing about the relationship between a node and its uncle, nothing about any two nodes that are not parent and child. It is a purely *local* constraint, and that weakness is exactly what makes heaps cheap to repair.

5.3.4 A concrete heap

Take

$$A = [16,\ 14,\ 10,\ 8,\ 7,\ 9,\ 3,\ 2,\ 4,\ 1].$$

Drawn as a tree it is

```
            16  (index 1)
          /    \
       14        10          (indices 2, 3)
      /  \      /  \
     8    7    9    3        (indices 4, 5, 6, 7)
    / \  /
   2  4 1                    (indices 8, 9, 10)
```

Now check the heap property, one parent at a time — every step written out, because this is the kind of check you must be able to do in your head by the end of the chapter:

- Node 1 holds 16; children $A[2] = 14$ and $A[3] = 10$. $16 \geq 14$ ✓, $16 \geq 10$ ✓
- Node 2 holds 14; children $A[4] = 8$ and $A[5] = 7$. $14 \geq 8$ ✓, $14 \geq 7$ ✓
- Node 3 holds 10; children $A[6] = 9$ and $A[7] = 3$. $10 \geq 9$ ✓, $10 \geq 3$ ✓
- Node 4 holds 8; children $A[8] = 2$ and $A[9] = 4$. $8 \geq 2$ ✓, $8 \geq 4$ ✓
- Node 5 holds 7; child $A[10] = 1$, and no right child since $2 \times 5 + 1 = 11 > 10$. $7 \geq 1$ ✓
- Nodes 6 through 10 have index $> \lfloor 10/2 \rfloor = 5$, so they have no children at all and nothing to check.

Every test passes, so A is a max-heap. **Figure 5.2** draws the same array both ways at once — as a tree, and as the flat array with arcs joining each parent to its two children — so that you can see the correspondence rather than take it on faith.

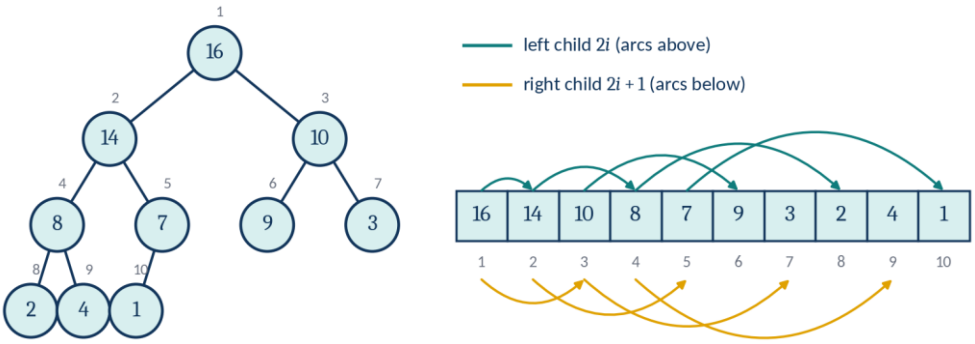


Figure 5.2 The same ten numbers seen twice. On the left, the nearly complete binary tree, with each node's array index printed above it. On the right, the array $A = [16,14,10,8,7,9,3,2,4,1]$, with an arc drawn from each node to each of its children — to the left child above the array, to the right child below it. The arcs are what the index arithmetic computes; no such arc is ever stored.

Look back at **Figure 5.1**: it draws the very slightly different array $A = [16,15,10,8,7,9,3,2,4,1]$. Check it yourself: $16 \geq 15$, $15 \geq 8$, $15 \geq 7$ — all fine. Both arrays are max-heaps, which makes a point worth making early: **a set of n numbers has many different valid heaps**, not one. The heap property is a loose constraint, not a total order. We use the version with 14 from here on because it is the array that the trace figures in Sections 5.5 and 5.6 are drawn from.

5.3.5 A heap is not sorted

This is the single most common misunderstanding about heaps, so we shall state it loudly.

> **COMMON TRAP.** A heap is **not** a sorted array. In $A = [16,14,10,8,7,9,3,2,4,1]$ the value 9 sits at index 6 and the value 8 sits at index 4, so a larger number appears *after* a smaller one. That is perfectly legal: 9 and 8 are not parent and child, so the heap property has nothing to say about them. The *only* global guarantee a max-heap makes is that $A[1]$ **is the maximum of the whole array**. Everything else is local.

Why is $A[1]$ the maximum? Because the heap property is transitive along any downward path. Take any node i. Then $A[\text{Parent}(i)] \geq A[i]$, and $A[\text{Parent}(\text{Parent}(i))] \geq A[\text{Parent}(i)]$, and so on. Following parents upward from i must reach the root after finitely many steps, because each step strictly decreases the index and indices are positive integers. Chaining the inequalities gives $A[1] \geq A[i]$. Since i was arbitrary, $A[1]$ dominates everything. ■

That one-line proof is the entire reason heaps exist. The maximum costs nothing to find: it is already sitting in slot 1.

5.3.6 The height of a heap is $\lfloor \lg n \rfloor$

THEOREM. A heap with n elements, being a nearly complete binary tree, has height $h = \lfloor \lg n \rfloor$.

Proof. Let the height be h. We show $2^h \leq n \leq 2^{h+1} - 1$, and then take logarithms.

The lower bound. A tree of height h has $h + 1$ levels, numbered 0 through h, and level h contains at least one node — otherwise the height would be less than h. Levels 0 through $h - 1$ are completely filled, by the definition of "nearly complete", and contain $2^0 + 2^1 + \cdots + 2^{h-1} = 2^h - 1$ nodes. Adding at least one node on level h gives

$$n \geq (2^h - 1) + 1 = 2^h.$$

The upper bound. Level h can hold at most 2^h nodes. So

$$n \leq (2^h - 1) + 2^h = 2^{h+1} - 1 < 2^{h+1}.$$

Putting the two together,

$$2^h \leq n < 2^{h+1}. \tag{5.4}$$

This chain of inequalities, Equation (5.4), pins n between two consecutive powers of two. Taking lg of all three parts — legitimate because lg is strictly increasing —

$$h \leq \lg n < h + 1,$$

which says precisely that h is the integer part of $\lg n$; that is, $h = \lfloor \lg n \rfloor$. ■

EXAMPLE. For $n = 10$: $\lg 10 \approx 3.32$, so $h = \lfloor 3.32 \rfloor = 3$. Check against the picture: the root is at depth 0, and the deepest nodes 8,9,10 are at depth 3. ✓ For $n = 1{,}000{,}000$: $\lg 10^6 \approx 19.93$, so $h = 19$. A heap of a million elements is **nineteen** levels deep. Any operation that walks one root-to-leaf path touches at most twenty nodes. That is why heap operations are fast.

5.3.7 Where the leaves live

THEOREM. In a heap of n elements, the leaves are exactly the nodes at indices $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$. Equivalently, the *internal* nodes — the ones with at least one child — are exactly $1, 2, \ldots, \lfloor n/2 \rfloor$.

Proof. Node i has a left child if and only if $\text{Left}(i) = 2i \leq n$, that is, if and only if $i \leq n/2$. Since i is an integer, that condition is $i \leq \lfloor n/2 \rfloor$. A node with no left child can have no right child either, since $2i + 1 > 2i > n$. So node i is internal exactly when $i \leq \lfloor n/2 \rfloor$, and is a leaf exactly when $i > \lfloor n/2 \rfloor$. ■

Two consequences that we shall use constantly:

1. **Just over half of a heap is leaves.** There are $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ of them.
2. **Every leaf is already a valid heap on its own.** A one-element tree has no parent-child pair to violate anything. This is what makes `BUILD-MAX-HEAP` able to skip the entire second half of the array, and it is half the reason that procedure runs in $O(n)$.

5.3.8 How many nodes sit at each height

For the $O(n)$ analysis in Section 5.5 we shall need one more count. Not the number of nodes at each *depth* (which is 2^d, easy), but the number at each *height* — where a node's height means the height of the subtree it roots, so leaves have height 0.

> **LEMMA.** In an n-element heap there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h.

The intuition is straightforward. The nodes of height 0 are the leaves, and we have just counted them: $\lceil n/2 \rceil = \lceil n/2^{0+1} \rceil$ ✓. Now strip the leaves away. What is left is again a nearly complete binary tree, with at most $\lfloor n/2 \rfloor$ nodes, and the nodes of height 1 in the original tree are precisely the leaves of this smaller tree — of which there are at most $\lceil \lfloor n/2 \rfloor / 2 \rceil \leq \lceil n/4 \rceil = \lceil n/2^{1+1} \rceil$ ✓. Repeating the strip-the-leaves argument gives the bound at every height, by induction on h.

The shape of that result is what matters, so let us say it in words:

> **IN PLAIN LANGUAGE.** A heap is overwhelmingly bottom-heavy. About half of all nodes are leaves, a quarter have height 1, an eighth have height 2, and so on. Only *one* node — the root — has the full height $\lfloor \lg n \rfloor$. So an operation whose cost is proportional to a node's height is cheap on almost every node, and expensive on almost none. Hold on to this sentence; it is the whole of Section 5.5.6.

5.3.9 All of it, in a program

Listing 5.1 puts every claim of this section into code you can run: the three navigation formulas, the height computed without touching a floating-point number, the leaf range, the node count at each height checked against the lemma's bound, and a final verification of the heap property. It uses the array [16,15,10,8,7,9,3,2,4,1] drawn in **Figure 5.1**.

Listing 5.1 Every structural fact about heaps from Section 5.3, computed and checked.

```
// Heap index arithmetic, height, leaves, and the node count at each height.
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
using namespace std;

int n;                                          // heap size, visible to the helpers

int parentOf(int i) { return i / 2; }           // integer division IS floor for i > 0
int leftOf  (int i) { return 2 * i; }
int rightOf (int i) { return 2 * i + 1; }

int heightOfHeap(int m) {                       // floor(lg m), no floating point anywhere
    int h = 0;
    while ((1 << (h + 1)) <= m) ++h;            // the largest h with 2^(h+1) <= m
    return h;
}

int subtreeHeight(int i) {                      // height of the subtree rooted at node i
    if (leftOf(i) > n) return 0;                // a leaf has height 0
    int best = subtreeHeight(leftOf(i));
    if (rightOf(i) <= n) best = max(best, subtreeHeight(rightOf(i)));
    return best + 1;
}

int main() {
    vector<int> A = {0, 16, 15, 10, 8, 7, 9, 3, 2, 4, 1};   // A[0] unused: 1-based
indexing
    n = 10;

    cout << "Array (1-based): ";
    for (int i = 1; i <= n; ++i) cout << A[i] << " ";
    cout << "\n\nindex  value  parent  left  right\n";
    for (int i = 1; i <= n; ++i) {
        cout << setw(5) << i << setw(7) << A[i];
        cout << setw(8); if (i > 1)            cout << parentOf(i); else cout << "-";
        cout << setw(6); if (leftOf(i)  <= n)  cout << leftOf(i);   else cout << "-";
        cout << setw(7); if (rightOf(i) <= n)  cout << rightOf(i);  else cout << "-";
        cout << "\n";
    }

    cout << "\nheight of the heap = floor(lg " << n << ") = " << heightOfHeap(n) << "\n";
    cout << "leaves occupy positions " << n / 2 + 1 << " .. " << n << "\n";
```

Listing 5.1 (Continued) *Every structural fact about heaps from Section 5.3, computed and checked.*

```
    cout << "\nNodes at each height h, against the bound ceil(n/2^(h+1)):\n";
    for (int h = 0; h <= heightOfHeap(n); ++h) {
        int actual = 0;
        for (int i = 1; i <= n; ++i) if (subtreeHeight(i) == h) ++actual;
        int bound = (n + (1 << (h + 1)) - 1) / (1 << (h + 1));  // ceil(n / 2^(h+1))
        cout << "  h = " << h << " : actual = " << actual
             << ", bound = " << bound << "\n";
    }

    bool ok = true;                             // verify the max-heap property everywhere
    for (int i = 2; i <= n; ++i) if (A[parentOf(i)] < A[i]) ok = false;
    cout << "\nmax-heap property holds: " << (ok ? "yes" : "no") << "\n";
    return 0;
}
```

Compiled with `g++ -std=c++17 -O2` and run, it prints:

```
Array (1-based): 16 15 10 8 7 9 3 2 4 1

index  value  parent  left  right
    1     16       -     2      3
    2     15       1     4      5
    3     10       1     6      7
    4      8       2     8      9
    5      7       2    10      -
    6      9       3     -      -
    7      3       3     -      -
    8      2       4     -      -
    9      4       4     -      -
   10      1       5     -      -

height of the heap = floor(lg 10) = 3
leaves occupy positions 6 .. 10

Nodes at each height h, against the bound ceil(n/2^(h+1)):
  h = 0 : actual = 5, bound = 5
  h = 1 : actual = 3, bound = 3
  h = 2 : actual = 1, bound = 2
  h = 3 : actual = 1, bound = 1

max-heap property holds: yes
```

Read the output against the theory. The height is 3, as proved. The leaves run from 6 to 10, as proved — and $\lfloor 10/2 \rfloor + 1 = 6$ ✓. Node 5 has a left child at index 10 but no right child, because $11 > 10$; the program prints a dash rather than inventing one, and getting that boundary right in your own code is the difference between a working heap and a crash. And the node counts by height, 5,3,1,1, never exceed the bound $\lceil n/2^{h+1} \rceil = 5,3,2,1$ — the bound is tight at heights 0,1,3 and slack at height 2, which is exactly what "at most" allows.

5.4 `MAX-HEAPIFY`: the Engine That Keeps the Heap Honest

5.4.1 The problem it solves

Everything a heap does — sorting, extracting, inserting — reduces to one repair operation. Here is the situation it is built for.

Imagine a node whose two subtrees are already perfectly good heaps, but whose own value is too small, so that it is beaten by one or both of its children. The heap property fails at exactly one place: this node. Everywhere else it holds.

```
          4        <-- too small: violates the heap property here
        /   \
      14     10    <-- but BOTH of these subtrees are already valid heaps
     /  \   /  \
    8    7 9    3
```

The value 4 is smaller than both its children, 14 and 10. But look below: the subtree rooted at 14 is a valid heap ($14 \geq 8$, $14 \geq 7$), and so is the subtree rooted at 10 ($10 \geq 9$, $10 \geq 3$). The damage is confined to one node.

> **DEFINITION.** `MAX-HEAPIFY`(A, i) (also written `HEAPIFY`). *Precondition:* the binary trees rooted at Left(i) and Right(i) are already max-heaps, but $A[i]$ may be smaller than one or both of its children. *Postcondition:* the subtree rooted at i is a max-heap, containing exactly the same multiset of values it contained before. The procedure achieves this by letting the value at $A[i]$ **float down** — repeatedly swapping it with the larger of its two children — until it reaches a position where it dominates both children, or becomes a leaf.

The word to fix in your mind is *sink*. The offending value sinks through the tree like a stone through water, and everything it passes floats up by one level.

5.4.2 The algorithm

Pseudocode 5.1 gives the procedure. You will also meet it under the shorter name `HEAPIFY`, which is how line 10 refers to it:

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
```

In plain English: look at three values — the node and its two children. Work out which of the three is the largest. If the node itself is largest, everything is already in order and there is nothing to do. Otherwise, swap the node with the largest child, and then repeat the whole procedure at the position the node has just moved into.

Three details in those ten lines are easy to get wrong, so we name them.

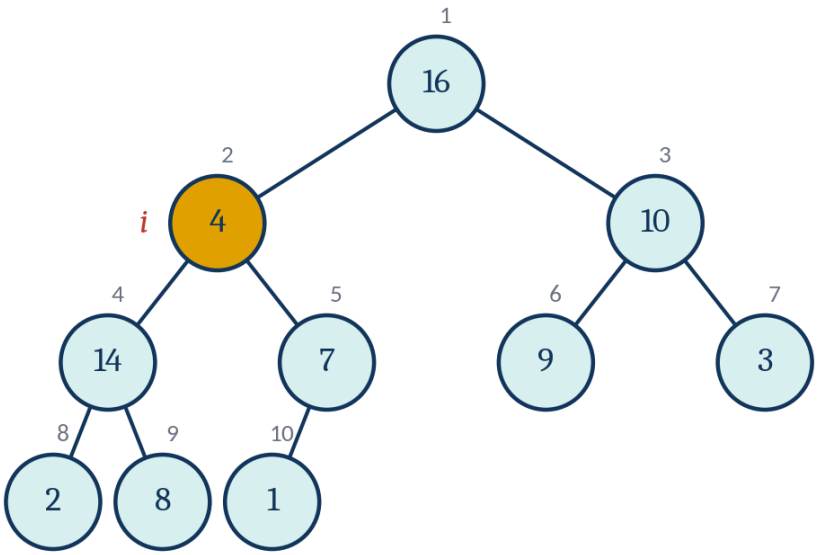
- **The guard `l ≤ heap-size`.** A node near the bottom may have one child or none. Reading $A[2i]$ without checking is reading past the end of the heap, which in C++ is undefined behavior and in practice is a bug that appears only on certain array lengths.
- **`heap-size`, not the array length.** These are two different numbers. The array may be long; the heap may occupy only a prefix of it. Heapsort in Section 5.6 depends entirely on that distinction, because it shrinks the heap while leaving the array alone.
- **Line 6 compares against `A[largest]`, not `A[i]`.** After line 5 may have set `largest ← l`, we want the largest of *three* values. Comparing the right child against $A[i]$ instead of against the current champion is the classic bug: it can pick the right child even when the left child is bigger, and the heap property then fails one level down.

5.4.3 A complete walkthrough

Take a concrete example. The array is

$$A = [16,\ 4,\ 10,\ 14,\ 7,\ 9,\ 3,\ 2,\ 8,\ 1], \qquad \text{heap-size} = 10,$$

and $A[2] = 4$ is the offender. We call `MAX-HEAPIFY(A, 2)`. **Figure 5.3** is the starting picture.



(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.

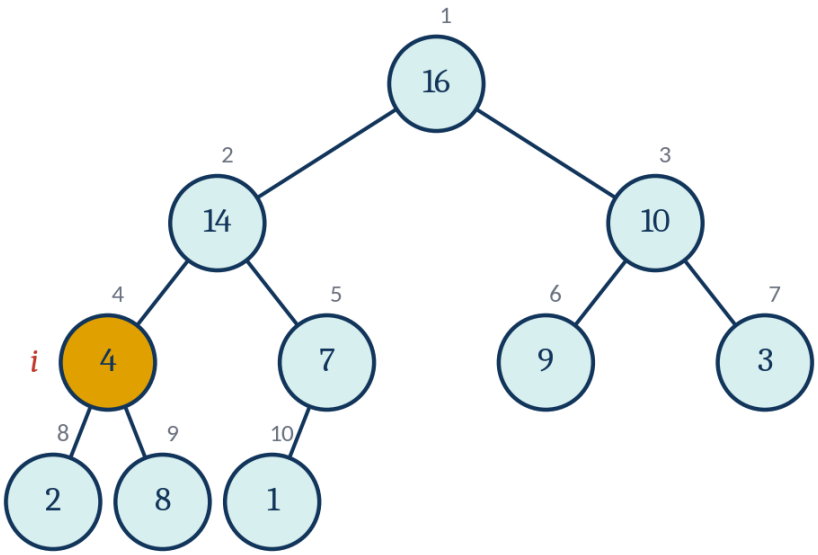
Step 1 — the call `MAX-HEAPIFY(A, 2)`**.** Work through the pseudocode one line at a time.

- Line 1: $l \leftarrow \text{Left}(2) = 2 \times 2 = 4$.
- Line 2: $r \leftarrow \text{Right}(2) = 2 \times 2 + 1 = 5$.
- Line 3: largest $\leftarrow 2$.
- Line 4: is $l = 4 \leq 10$? Yes. Is $A[4] = 14 > A[2] = 4$? Yes.
- Line 5: so largest $\leftarrow 4$.
- Line 6: is $r = 5 \leq 10$? Yes. Is $A[5] = 7 > A[\text{largest}] = A[4] = 14$? No.
- Line 7 is skipped; largest stays 4.
- Line 8: is largest $= 4 \neq i = 2$? Yes.
- Line 9: swap $A[2] \leftrightarrow A[4]$. The array becomes

$$A = [16,\ \mathbf{14},\ 10,\ \mathbf{4},\ 7,\ 9,\ 3,\ 2,\ 8,\ 1].$$

- Line 10: recurse on `MAX-HEAPIFY(A, 4)`.

Figure 5.4 shows the state now. The 4 has sunk one level, to index 4, and the 14 has risen to index 2.



(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.

Step 2 — the call `MAX-HEAPIFY(A, 4)`.

- $l \leftarrow 2 \times 4 = 8, r \leftarrow 2 \times 4 + 1 = 9$, largest $\leftarrow 4$.
- Is $8 \leq 10$? Yes. Is $A[8] = 2 > A[4] = 4$? No. largest stays 4.
- Is $9 \leq 10$? Yes. Is $A[9] = 8 > A[\text{largest}] = A[4] = 4$? Yes, so largest $\leftarrow 9$.
- largest $= 9 \neq 4$, so swap $A[4] \leftrightarrow A[9]$:

$$A = [16,\ 14,\ 10,\ \mathbf{8},\ 7,\ 9,\ 3,\ 2,\ \mathbf{4},\ 1].$$

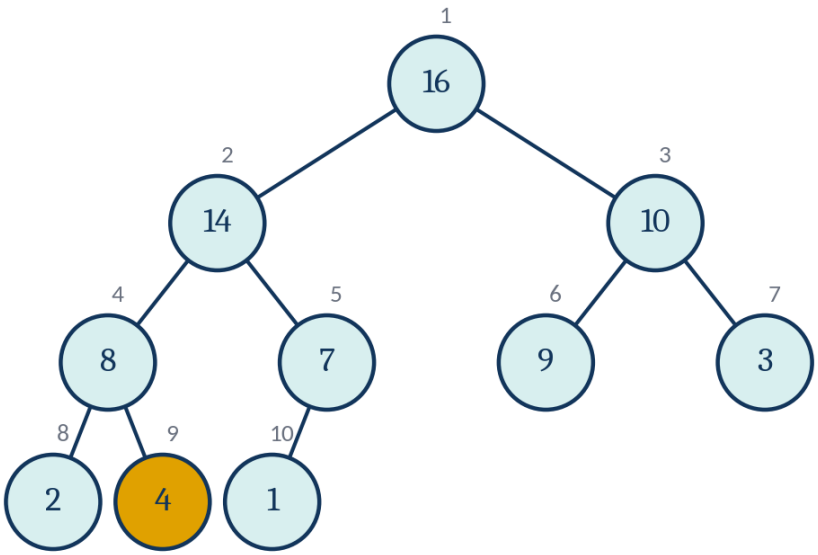
- Recurse on `MAX-HEAPIFY(A, 9)`.

Note that this step is where the "compare against `A[largest]`" rule earned its keep. Had line 6 compared $A[9] = 8$ against $A[i] = 4$ we would have got the same answer here by luck — but had the children been $A[8] = 6$ and $A[9] = 5$, the buggy version would have moved 5 up instead of 6, leaving $6 > 5$ one level down and a broken heap.

Step 3 — the call `MAX-HEAPIFY(A, 9)`.

- $l \leftarrow 2 \times 9 = 18, r \leftarrow 19$, largest $\leftarrow 9$.
- Is $18 \leq 10$? **No.** The test on line 4 fails at the first condition and the second condition is never evaluated. Skip.
- Is $19 \leq 10$? No. Skip.
- largest $= 9 = i$, so line 8 fails, no swap happens, and the procedure returns.

Node 9 is a leaf, and a leaf is trivially a heap. The recursion has bottomed out. **Figure 5.5** is the finished state.



(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.

The final array is

$$A = [16,\ 14,\ 10,\ 8,\ 7,\ 9,\ 3,\ 2,\ 4,\ 1],$$

which is the heap we checked by hand in Section 5.3.4. The value 4 sank from position 2 to position 4 to position 9 — **three positions along a single downward path**, and nothing anywhere else in the tree was touched.

5.4.4 Why it is correct

The procedure is recursive, so the proof is by induction on the height of node i.

Base case. If i is a leaf, it has no children, both guards fail, largest $= i$, and the procedure returns immediately leaving A unchanged. A single node is a max-heap. ✓

Inductive step. Suppose the procedure is correct on every node of height less than h, and let i have height $h \geq 1$. By the precondition, the subtrees at Left(i) and Right(i) are max-heaps. Lines 1–7 compute largest so that A[largest] is the greatest of $A[i]$, A[Left(i)] and A[Right(i)] — where missing children are simply not considered. Two cases:

- If largest $= i$, then $A[i]$ already dominates both children. Since both subtrees are max-heaps, every node in them is dominated by its own parent, and the roots of those subtrees are dominated by $A[i]$. So the whole subtree at i is a max-heap and we are done, correctly, without doing anything.
- If largest $\neq i$ — say largest $=$ Left(i) — then after the swap, $A[i]$ holds the largest of the three values, so it dominates A[Right(i)] (unchanged, still a heap root) and A[Left(i)] (which now holds the old $A[i]$). The right subtree is untouched and still a max-heap. The left subtree now has a possibly-too-small root but two valid heap children — which is exactly the precondition of `MAX-HEAPIFY` at Left(i). That node has height at most $h - 1$, so by the inductive hypothesis the recursive call repairs it. The whole subtree at i is then a max-heap. ✓

The symmetric case largest = Right(i) is identical with left and right exchanged. And no element is ever created or destroyed — every line either reads a value or exchanges two of them — so the multiset of values is preserved. ■

5.4.5 The running time

Each call does a fixed amount of work: two index computations, at most two comparisons, at most one exchange. That is $\Theta(1)$. Then it may recurse on one child.

So the total cost is $\Theta(1)$ multiplied by the number of levels the value descends — and since each call descends exactly one level, that is at most the height of node i. With the height of the whole heap being $\lfloor \lg n \rfloor$ (Section 5.3.6), we have immediately:

$$T(n) = O(\lg n).$$

That is the honest one-line argument, and it is completely sufficient. But the recurrence version is worth doing properly too, because it is a nice exercise in the Master Theorem from Chapter 3.

Writing the recurrence. When `MAX-HEAPIFY` recurses, it recurses on one of the two subtrees of node i. How big can that subtree be, as a fraction of the subtree at i?

Suppose the subtree at i has m nodes. If the tree below i were perfectly complete, each child subtree would hold $(m-1)/2 < m/2$ nodes and the answer would be a clean half. But the tree need not be complete: the *last* level may be partly filled, and it fills **from the left**. The worst case for our purposes is therefore a subtree whose bottom level is exactly half full — because then the left child's subtree is as large as it can be relative to the right child's.

Work it out. Say the subtree at i has height h, its levels 0 through $h-1$ (counted from node i) are full, and its bottom level h is exactly half full. Count the nodes in three steps.

- *The whole subtree.* Levels 0 through $h-1$ hold $1+2+\cdots+2^{h-1} = 2^h - 1$ nodes. A full bottom level would hold 2^h nodes; half of it holds 2^{h-1}. So $m = (2^h - 1) + 2^{h-1}$.
- *The left child's subtree.* Each of the levels 1 through $h-1$ gives exactly half of its nodes to the left side, which is $1+2+\cdots+2^{h-2} = 2^{h-1} - 1$ nodes. And because the bottom level fills from the left, **all** 2^{h-1} of its nodes belong to the left side too. Total: $(2^{h-1} - 1) + 2^{h-1} = 2^h - 1$.
- *The right child's subtree.* It gets only its half of the upper levels, $2^{h-1} - 1$ nodes, and nothing from the bottom level. (Check: $1 + (2^h - 1) + (2^{h-1} - 1) = 2^h + 2^{h-1} - 1 = m$ ✓.)

The left side's share of the whole subtree is therefore

$$\frac{2^h - 1}{2^h - 1 + 2^{h-1}} = \frac{2 \cdot 2^{h-1} - 1}{3 \cdot 2^{h-1} - 1} < \frac{2}{3}.$$

Why is the last step true? Write $x = 2^{h-1}$. The claim $\frac{2x-1}{3x-1} < \frac{2}{3}$ becomes, after multiplying both sides by the positive numbers 3 and $3x-1$, the claim $3(2x-1) < 2(3x-1)$, that is, $6x - 3 < 6x - 2$, which is always true. As h grows the share creeps up toward $2/3$ but never reaches it. Any other filling of the bottom level gives the left side a smaller share: a less-than-half-full bottom level gives the left side fewer nodes, and a more-than-half-full one gives the extra nodes to the right side.

So a child's subtree never holds more than $2/3$ of the nodes of its parent's subtree, and the recurrence is

$$T(n) \leq T(2n/3) + \Theta(1). \tag{5.5}$$

Solving it with the Master Theorem. In Equation (5.5) we have $a = 1$, $b = 3/2$, and $f(n) = \Theta(1) = \Theta(n^0)$. Compute the watershed exponent:

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1,$$

because $\log_b 1 = 0$ for any base b. So $f(n) = \Theta(1) = \Theta(n^{\log_b a} \lg^0 n)$, which is Case 2 of the Master Theorem with $k = 0$. Case 2 gives

$$T(n) = \Theta(n^{\log_b a} \lg^{k+1} n) = \Theta(n^0 \lg^1 n) = \Theta(\lg n).$$

Since the recurrence was an inequality ($\leq$), the conclusion is an upper bound:

$$T(n) = O(\lg n). \qquad \blacksquare$$

> **IN PLAIN LANGUAGE.** `MAX-HEAPIFY` walks down one path from the node it was given to, at worst, a leaf. A heap of n elements is only $\lfloor \lg n \rfloor$ levels tall, so that walk is short. For a million elements the walk is at most nineteen steps. For a billion elements it is at most twenty-nine. That is the whole of the $O(\lg n)$ claim; the recurrence above is simply the same fact written in the language of Chapter 3.

5.4.6 `MAX-HEAPIFY` in C++

Listing 5.2 implements the procedure and prints every decision it takes, so that you can compare the machine's trace against the hand trace of Section 5.4.3 line by line.

Listing 5.2 `MAX-HEAPIFY` in C++, narrating each of its own decisions.

```
// MAX-HEAPIFY, with every step of the sinking printed.
#include <iostream>
#include <vector>
#include <string>
using namespace std;

vector<int> A;          // A[0] is unused so that the index arithmetic is the textbook one
int heapSize;

void show(const string& label) {
    cout << label;
    for (int i = 1; i <= heapSize; ++i) cout << A[i] << (i < heapSize ? " " : "");
    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;

    cout << "  MAX-HEAPIFY(A, " << i << "): A[" << i << "]=" << A[i];
    if (l <= heapSize) cout << ", left A[" << l << "]=" << A[l]; else cout << ", no left
child";
    if (r <= heapSize) cout << ", right A[" << r << "]=" << A[r]; else cout << ", no right
child";
    cout << " -> largest is index " << largest << "\n";

    if (largest != i) {
        swap(A[i], A[largest]);
        show("    swap, giving A = [");
        maxHeapify(largest);                    // the value keeps sinking
    } else {
        cout << "    node " << i << " already dominates its children: stop\n";
    }
}

int main() {
    A = {0, 16, 4, 10, 14, 7, 9, 3, 2, 8, 1};   // A[2] = 4 breaks the heap property
    heapSize = 10;
    show("start:  A = [");
    cout << "\ncalling MAX-HEAPIFY(A, 2)\n";
    maxHeapify(2);
    show("\nresult: A = [");
```

Listing 5.2 (Continued) *MAX-HEAPIFY in C++, narrating each of its own decisions.*

```
    bool ok = true;
    for (int i = 2; i <= heapSize; ++i) if (A[i / 2] < A[i]) ok = false;
    cout << "valid max-heap: " << (ok ? "yes" : "no") << "\n";
    return 0;
}
```

The program prints:

```
start:  A = [16 4 10 14 7 9 3 2 8 1]

calling MAX-HEAPIFY(A, 2)
  MAX-HEAPIFY(A, 2): A[2]=4, left A[4]=14, right A[5]=7 -> largest is index 4
    swap, giving A = [16 14 10 4 7 9 3 2 8 1]
  MAX-HEAPIFY(A, 4): A[4]=4, left A[8]=2, right A[9]=8 -> largest is index 9
    swap, giving A = [16 14 10 8 7 9 3 2 4 1]
  MAX-HEAPIFY(A, 9): A[9]=4, no left child, no right child -> largest is index 9
    node 9 already dominates its children: stop

result: A = [16 14 10 8 7 9 3 2 4 1]
valid max-heap: yes
```

Three calls, three levels, exactly as the hand trace predicted — and the resulting array matches **Figure 5.5** element for element. Note the short-circuit in `l <= heapSize && A[l] > A[largest]`: when `l` is out of range C++ never evaluates `A[l]`, which is why the third call prints "no left child" instead of crashing.

> **TRAP.** A common student version writes `if (A[l] > A[largest] && l <= heapSize)`. This is a real bug. C++ evaluates the left operand first, so `A[l]` is read *before* the bounds test, and the read is out of range. It will often appear to work — the memory just past the vector usually contains something harmless — and then fail on a different input or a different compiler. Order matters.

5.5 `BUILD-MAX-HEAP`: the Surprising Linear-Time Construction

5.5.1 The algorithm

We can repair a heap. Now we need to *make* one, from an array in arbitrary order. **Pseudocode 5.2** does it.

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)
```

Three lines. Call `MAX-HEAPIFY` on every internal node, from the last one to the first.

Two choices in those three lines carry the whole weight of the procedure, and each needs an explanation.

Why start at $\lfloor n/2 \rfloor$? Because by the theorem of Section 5.3.7, the nodes $\lfloor n/2 \rfloor + 1$ through n are precisely the leaves, and a leaf is already a valid one-element heap. There is nothing to repair. Calling `MAX-HEAPIFY` on them would be correct but pointless — and, as Section 5.5.6 will show, skipping them is not a minor saving: it is *half the array*.

Why count downward? Because `MAX-HEAPIFY`(A, i) has a precondition: both subtrees of i must already be heaps. The children of node i sit at indices $2i$ and $2i + 1$, both strictly greater than i. So if we process indices in decreasing order, every child is finished before its parent is started, and the precondition is automatically satisfied at every call. Counting *upward* would call `MAX-HEAPIFY` on a node whose subtrees are still in disarray, the precondition would be violated, and the result would simply be wrong.

5.5.2 Correctness, by loop invariant

> **LOOP INVARIANT.** At the start of each iteration of the `for` loop of line 2, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.

Initialization. Before the first iteration, $i = \lfloor n/2 \rfloor$. The nodes $\lfloor n/2 \rfloor + 1, \ldots, n$ are leaves, and every leaf is a one-element max-heap. The invariant holds. ✓

Maintenance. Assume the invariant holds at the start of an iteration with counter i. The children of node i are at $2i$ and $2i+1$, and both of those indices are greater than i, so by the invariant both are roots of max-heaps. That is exactly `MAX-HEAPIFY`'s precondition, so the call on line 3 is legitimate and — by the correctness proof of Section 5.4.4 — it makes node i the root of a max-heap. It does not disturb anything outside the subtree at i, so nodes $i+1, \ldots, n$ remain heap roots too. The loop then decrements i, and at the start of the next iteration nodes $i, i+1, \ldots, n$ are all heap roots, which is the invariant with i replaced by $i-1$. ✓

Termination. The loop ends when $i = 0$. Substituting into the invariant: every node $1, 2, \ldots, n$ is the root of a max-heap. In particular node 1 is, and the subtree rooted at node 1 is the whole array. So $A[1 \ldots n]$ is a max-heap. ✓ ■

5.5.3 A complete walkthrough

Take this input array,

$$A = [4,\ 1,\ 3,\ 2,\ 16,\ 9,\ 10,\ 14,\ 8,\ 7],$$

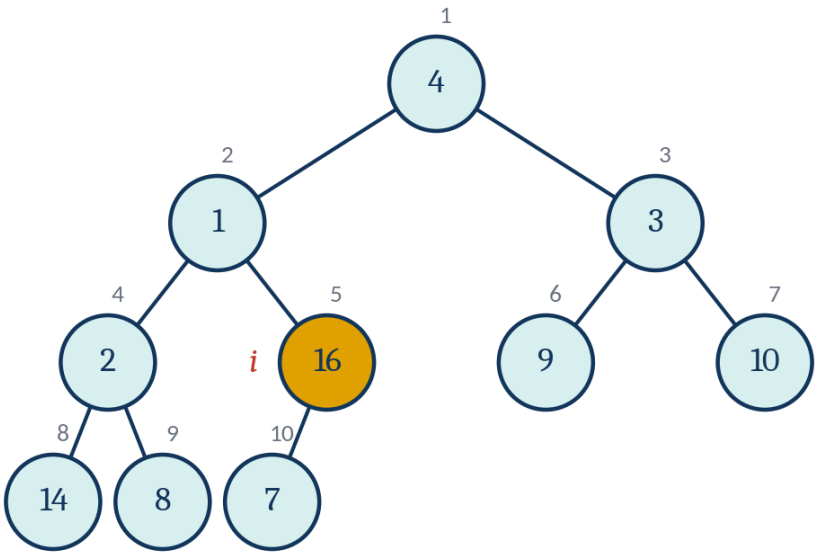
shown in **Figure 5.6**. There is no order in it at all.

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

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.

Here $n = 10$, so $\lfloor n/2 \rfloor = 5$ and the loop runs for $i = 5,4,3,2,1$. Nodes 6 through 10 — holding 9,10,14,8,7 — are leaves and are skipped entirely.

Iteration $i = 5$. Figure 5.7 is the state.



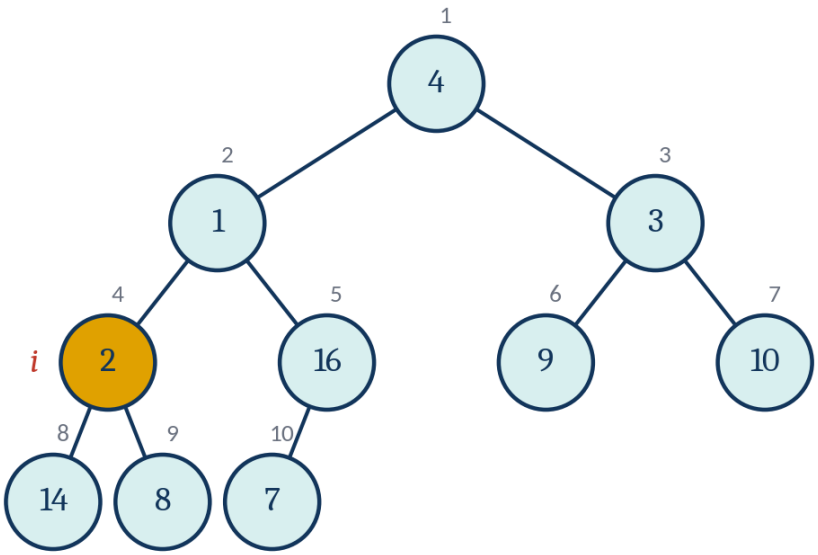
(a)

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.

- $A[5] = 16$. Left child $A[10] = 7$. Right child would be $A[11]$, which does not exist since $11 > 10$.
- Is $16 > 7$? Yes, so node 5 already dominates its only child.
- largest $= 5$: **no swap**. The array is unchanged:

$$A = [4,\ 1,\ 3,\ 2,\ 16,\ 9,\ 10,\ 14,\ 8,\ 7].$$

Iteration $i = 4$. Figure 5.8.



(b)

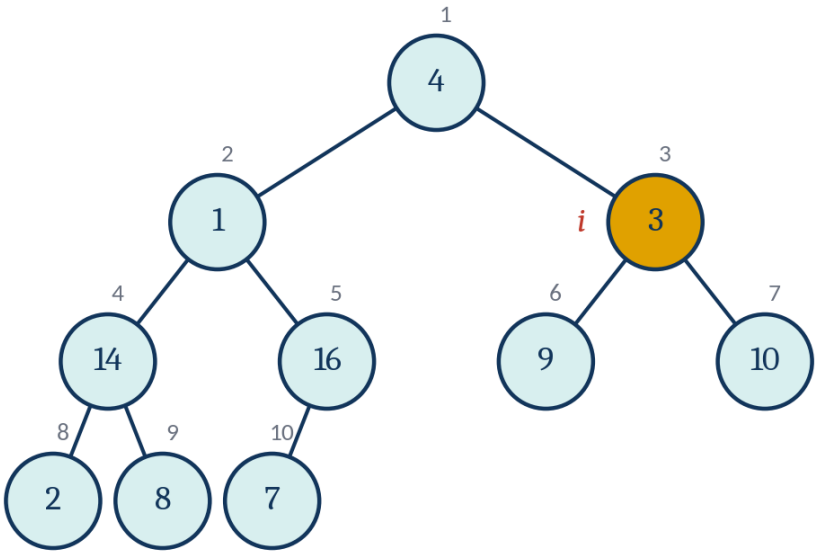
Figure 5.8 Step (b): $i = 4$**.** Node 4, shaded, holds 2, and its children $A[8] = 14$ and $A[9] = 8$ are both larger. The value 2 must sink.

- $A[4] = 2$. Children $A[8] = 14$, $A[9] = 8$.
- Largest of $\{2,14,8\}$ is 14, at index 8. Swap $A[4] \leftrightarrow A[8]$:

$$A = [4,\ 1,\ 3,\ \mathbf{14},\ 16,\ 9,\ 10,\ \mathbf{2},\ 8,\ 7].$$

- Recurse at index 8: $2 \times 8 = 16 > 10$, so node 8 is a leaf. Stop.

Iteration $i = 3$. Figure 5.9.



(c)

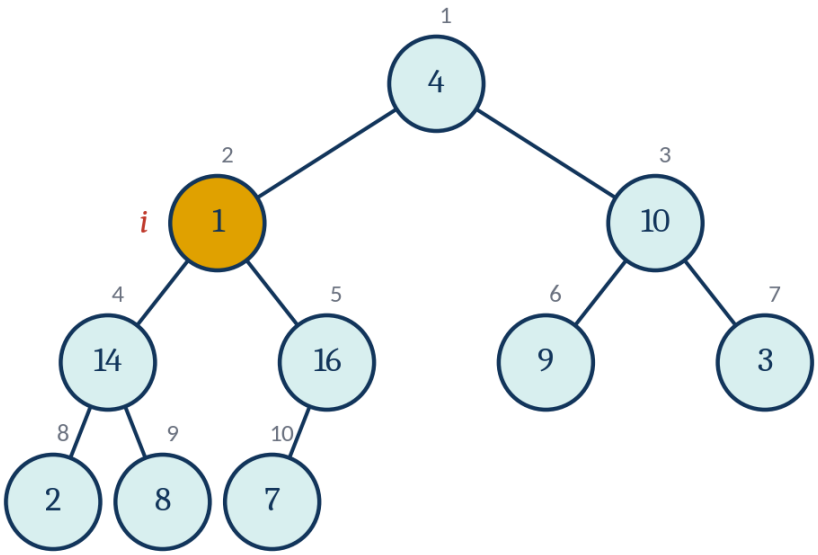
Figure 5.9 Step (c): $i = 3$. Node 3, shaded, holds 3; its children are $A[6] = 9$ and $A[7] = 10$. The right child is the larger, so the exchange goes to the right.

- $A[3] = 3$. Children $A[6] = 9$, $A[7] = 10$.
- Largest of $\{3,9,10\}$ is 10, at index 7. Swap $A[3] \leftrightarrow A[7]$:

$$A = [4,\ 1,\ \mathbf{10},\ 14,\ 16,\ 9,\ \mathbf{3},\ 2,\ 8,\ 7].$$

- Recurse at index 7: $2 \times 7 = 14 > 10$, a leaf. Stop.

Iteration $i = 2$. Figure 5.10.



(d)

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.

- $A[2] = 1$. Children $A[4] = 14$, $A[5] = 16$.
- Largest of $\{1,14,16\}$ is 16, at index 5. Swap $A[2] \leftrightarrow A[5]$:

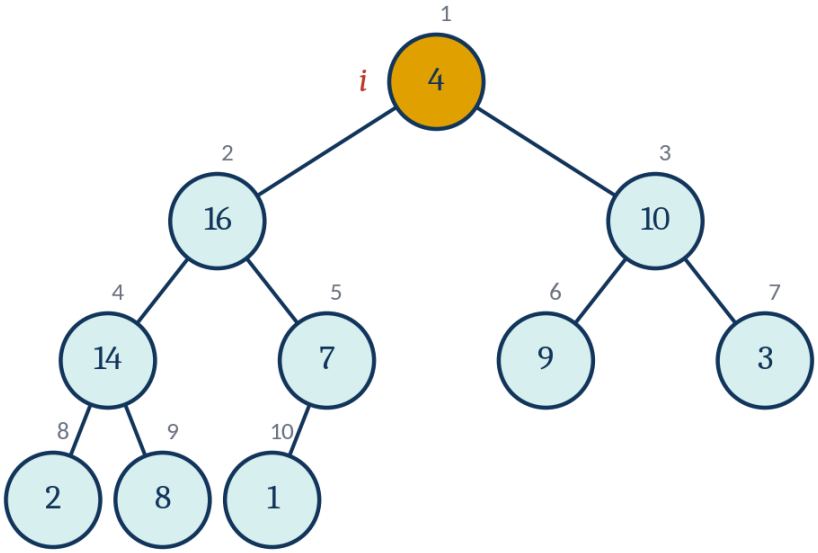
$$A = [4,\ \mathbf{16},\ 10,\ 14,\ \mathbf{1},\ 9,\ 3,\ 2,\ 8,\ 7].$$

- Recurse at index 5. $A[5] = 1$, left child $A[10] = 7$, no right child ($11 > 10$).
- Largest of $\{1,7\}$ is 7. Swap $A[5] \leftrightarrow A[10]$:

$$A = [4,\ 16,\ 10,\ 14,\ \mathbf{7},\ 9,\ 3,\ 2,\ 8,\ \mathbf{1}].$$

- Recurse at index 10: a leaf. Stop.

Iteration $i = 1$. Figure 5.11.



(e)

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.

- $A[1] = 4$. Children $A[2] = 16$, $A[3] = 10$.
- Largest of $\{4,16,10\}$ is 16, at index 2. Swap $A[1] \leftrightarrow A[2]$:

$$A = [\mathbf{16},\ \mathbf{4},\ 10,\ 14,\ 7,\ 9,\ 3,\ 2,\ 8,\ 1].$$

- Recurse at index 2. $A[2] = 4$, children $A[4] = 14$, $A[5] = 7$. Largest is 14 at index 4. Swap:

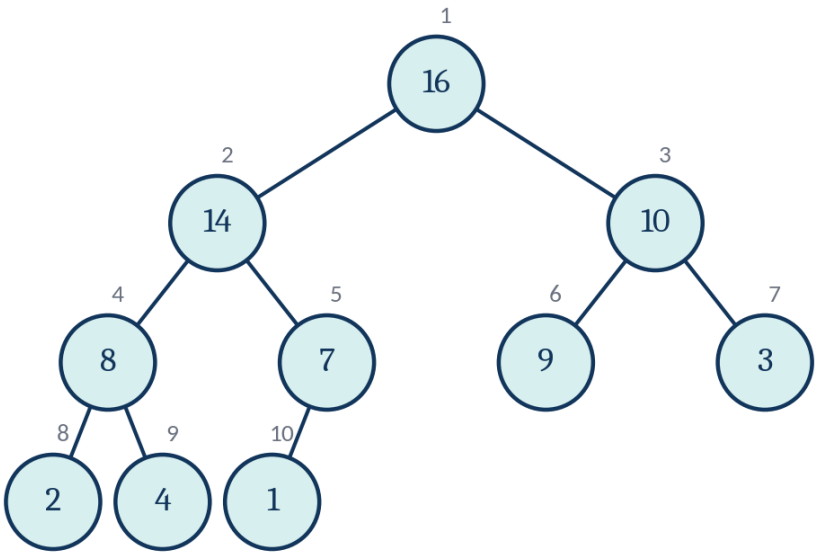
$$A = [16,\ \mathbf{14},\ 10,\ \mathbf{4},\ 7,\ 9,\ 3,\ 2,\ 8,\ 1].$$

- Recurse at index 4. $A[4] = 4$, children $A[8] = 2$, $A[9] = 8$. Largest is 8 at index 9. Swap:

$$A = [16,\ 14,\ 10,\ \mathbf{8},\ 7,\ 9,\ 3,\ 2,\ \mathbf{4},\ 1].$$

- Recurse at index 9: a leaf. Stop.

The loop is finished. **Figure 5.12** is the result.



(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.

Observe that the last iteration alone caused three exchanges — the value 4 fell from the root all the way to a leaf — while the first two iterations caused at most one apiece. That asymmetry is the seed of the analysis to come.

5.5.4 The loose bound: $O(n\lg n)$

Here is the analysis a careless reader would produce.

The loop runs $\lfloor n/2 \rfloor$ times. Each iteration calls `MAX-HEAPIFY`, which costs $O(\lg n)$. Therefore `BUILD-MAX-HEAP` costs $\lfloor n/2 \rfloor \times O(\lg n) = O(n\lg n)$.

Every step of that is **true**. The bound is correct. It is simply not *tight*: the real cost is much smaller, and $O(n\lg n)$ is an honest upper bound the way "this room holds fewer than a million people" is an honest upper bound on a classroom.

Where does the slack come in? In the phrase "each iteration costs $O(\lg n)$". `MAX-HEAPIFY`(A, i) does not cost $\lg n$; it costs the *height of node* i. For the root that is indeed $\lg n$. For a node just above the leaves it is 1. And, as Section 5.3.8 told us, the overwhelming majority of nodes are just above the leaves.

5.5.5 The tight bound: $O(n)$, derived step by step

We now do the analysis properly. Nothing here is harder than Chapter 3's summations; it just has to be done carefully.

Step 1 — the cost of one call, in terms of height. A call to `MAX-HEAPIFY` on a node of height h walks down at most h levels, doing $\Theta(1)$ work per level. So its cost is $O(h)$.

Step 2 — how many nodes have each height. By the lemma of Section 5.3.8, the number of nodes of height h is at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$.

Step 3 — the heights that occur. Heights run from 0 (the leaves) up to $\lfloor \lg n \rfloor$ (the root).

Step 4 — put them together. Summing the cost over all nodes, grouped by height:

$$T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} \underbrace{\left\lceil \frac{n}{2^{h+1}} \right\rceil}_{\text{how many nodes of height } h} \times \underbrace{O(h)}_{\text{cost of each}} . \tag{5.6}$$

Step 5 — get rid of the ceiling in Equation (5.6). We use a small fact: for any real $x \geq \frac{1}{2}$, $\lceil x \rceil \leq 2x$. (If $1/2 \leq x \leq 1$, then $\lceil x \rceil = 1 \leq 2x$; if $x > 1$, then $\lceil x \rceil < x + 1 < 2x$.) Every height h that occurs satisfies $h \leq \lg n$, that is, $2^h \leq n$, so $x = n/2^{h+1} \geq \frac{1}{2}$ and therefore $\lceil n/2^{h+1} \rceil \leq 2 \cdot n/2^{h+1} = n/2^h$. So, pulling the constant out of the O:

$$T(n) = O\left(\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} \cdot h\right) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right).$$

Step 6 — extend the sum to infinity. Every term $h/2^h$ is non-negative, so adding more terms only increases the sum. Replacing the finite upper limit by ∞ therefore preserves the inequality — and buys us a sum with a closed form:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h}.$$

Step 7 — evaluate the infinite sum. We prove in Section 5.5.6 that it equals exactly 2; the result is Equation (5.8).

Step 8 — conclude.

$$T(n) = O(n \cdot 2) = O(n). \qquad \blacksquare$$

And since `BUILD-MAX-HEAP` must at minimum look at all n elements, $T(n) = \Omega(n)$ too, so in fact $T(n) = \Theta(n)$.

> **WHY THIS IS SURPRISING, AND WHY IT IS TRUE.** Building a heap costs *less* than sorting it — less, even, than one pass of `MAX-HEAPIFY` per element would suggest. The reason is a simple counting fact: **the expensive calls are rare and the cheap calls are common.** Exactly one node has height $\lg n$. Two have height $\lg n - 1$. Half of all nodes have height 0 and cost nothing at all. Multiply "few and expensive" against "many and cheap" and the sum converges instead of growing. This is the clearest example in the whole course of why *tight* analysis matters: the loose bound $O(n \lg n)$ would have made us reject an $O(n)$ algorithm as too slow.

5.5.6 The identity $\sum_{h=0}^{\infty} h/2^h = 2$, proved

We owe a proof of Step 7. It comes from differentiating the geometric series — a technique worth keeping, because it turns up again in probability and in amortized analysis (Chapter 14).

Start from the geometric series. For any real x with $|x| < 1$,

$$\sum_{i=0}^{\infty} x^i = 1 + x + x^2 + x^3 + \cdots = \frac{1}{1-x}.$$

This is the formula from Chapter 3; it converges precisely because $|x| < 1$.

Differentiate both sides with respect to x. Within the radius of convergence a power series may be differentiated term by term. On the left,

$$\frac{d}{dx}\sum_{i=0}^{\infty} x^i \;=\; \sum_{i=0}^{\infty} \frac{d}{dx} x^i \;=\; \sum_{i=1}^{\infty} i\; x^{i-1}$$

— the $i = 0$ term is the constant 1, whose derivative is 0, which is why the index starts at 1. On the right, by the quotient or chain rule,

$$\frac{d}{dx}\left(\frac{1}{1-x}\right) = \frac{d}{dx}(1-x)^{-1} = -1 \cdot (1-x)^{-2} \cdot (-1) = \frac{1}{(1-x)^2}.$$

So

$$\sum_{i=1}^{\infty} i\; x^{i-1} \;=\; \frac{1}{(1-x)^2}.$$

Multiply both sides by x. This shifts the exponent up by one, which is what we want:

$$\sum_{i=1}^{\infty} i\; x^{i} \;=\; \frac{x}{(1-x)^2}. \tag{5.7}$$

Substitute $x = \frac{1}{2}$ into Equation (5.7)**.** It is legal, since $|1/2| < 1$:

$$\sum_{i=1}^{\infty} \frac{i}{2^i} \;=\; \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} \;=\; \frac{\frac{1}{2}}{\left(\frac{1}{2}\right)^2} \;=\; \frac{\frac{1}{2}}{\frac{1}{4}} \;=\; 2.$$

Finally, the term at $i = 0$ is $0/2^0 = 0$, so including it changes nothing:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} \;=\; 2. \qquad \blacksquare \tag{5.8}$$

If you prefer to see the number rather than the algebra, add the first few terms by hand:

$$\begin{aligned} &\frac{0}{1} + \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \frac{5}{32} + \frac{6}{64} + \frac{7}{128} \\ &= 0 + 0.5 + 0.5 + 0.375 + 0.25 + 0.15625 + 0.09375 + 0.0546875 \\ &= 1.9296875, \end{aligned}$$

already within four percent of 2 after eight terms, and climbing toward it.

5.5.7 Seeing the linear cost on a real machine

Theory is one thing; **Listing 5.3** is the check. It first traces `BUILD-MAX-HEAP` on the array of Section 5.5.3, printing every exchange, and then runs the procedure on random arrays of growing size and reports the number of exchanges **divided by** n. If the cost were $\Theta(n\lg n)$ that ratio would climb like $\lg n$. If it is $\Theta(n)$, the ratio will sit on a constant.

Listing 5.3 `BUILD-MAX-HEAP`, traced on the array of Section 5.5.3 and then measured on random arrays up to ten million elements.

```
// BUILD-MAX-HEAP: the trace on the array of Section 5.5.3, then evidence that it costs
O(n).
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <random>
using namespace std;

vector<int> A;
int heapSize;
long long swaps;                                  // how many exchanges MAX-HEAPIFY performs

void show(const string& label) {
    cout << label;
    for (int i = 1; i <= heapSize; ++i) cout << A[i] << (i < heapSize ? " " : "");
    cout << "]\n";
}

void maxHeapify(int i, bool verbose) {
    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]);
        ++swaps;
        if (verbose) { cout << "      swap A[" << i << "] with A[" << largest << "] -> A =
[";
                       for (int k = 1; k <= heapSize; ++k) cout << A[k] << (k < heapSize ?
" " : "");
                       cout << "]\n"; }
        maxHeapify(largest, verbose);
    }
}

void buildMaxHeap(bool verbose) {
    for (int i = heapSize / 2; i >= 1; --i) {
        if (verbose) cout << "  i = " << i << ", A[" << i << "] = " << A[i] << "\n";
        maxHeapify(i, verbose);
        if (verbose) { cout << "      after this call: A = [";
                       for (int k = 1; k <= heapSize; ++k) cout << A[k] << (k < heapSize ?
" " : "");
                       cout << "]\n"; }
    }
}
```

Listing 5.3 (Continued) *BUILD-MAX-HEAP, traced on the array of Section 5.5.3 and then measured on random arrays up to ten million elements.*

```
int main() {
    A = {0, 4, 1, 3, 2, 16, 9, 10, 14, 8, 7};    // the input array of Section 5.5.3
    heapSize = 10; swaps = 0;
    show("input:  A = [");
    cout << "floor(n/2) = " << heapSize / 2 << ", so the loop runs i = 5, 4, 3, 2, 1\n\n";
    buildMaxHeap(true);
    show("\nresult: A = [");
    cout << "exchanges performed: " << swaps << "\n";

    // Now the empirical check that the total cost grows like n, not like n lg n.
    cout << "\n         n     exchanges   exchanges/n   (n lg n)/n = lg n\n";
    mt19937 rng(12345);
    for (int n : {1000, 10000, 100000, 1000000, 10000000}) {
        A.assign(n + 1, 0);
        for (int i = 1; i <= n; ++i) A[i] = (int)(rng() % 1000000000u);
        heapSize = n; swaps = 0;
        buildMaxHeap(false);
        double lg = 0; { int m = n; while (m > 1) { m >>= 1; lg += 1; } }
        cout << setw(10) << n << setw(14) << swaps
             << setw(14) << fixed << setprecision(3) << (double)swaps / n
             << setw(18) << lg << "\n";
    }
    return 0;
}
```

The output:

```
input:  A = [4 1 3 2 16 9 10 14 8 7]
floor(n/2) = 5, so the loop runs i = 5, 4, 3, 2, 1

  i = 5, A[5] = 16
      after this call: A = [4 1 3 2 16 9 10 14 8 7]
  i = 4, A[4] = 2
      swap A[4] with A[8] -> A = [4 1 3 14 16 9 10 2 8 7]
      after this call: A = [4 1 3 14 16 9 10 2 8 7]
  i = 3, A[3] = 3
      swap A[3] with A[7] -> A = [4 1 10 14 16 9 3 2 8 7]
      after this call: A = [4 1 10 14 16 9 3 2 8 7]
  i = 2, A[2] = 1
      swap A[2] with A[5] -> A = [4 16 10 14 1 9 3 2 8 7]
      swap A[5] with A[10] -> A = [4 16 10 14 7 9 3 2 8 1]
      after this call: A = [4 16 10 14 7 9 3 2 8 1]
  i = 1, A[1] = 4
      swap A[1] with A[2] -> A = [16 4 10 14 7 9 3 2 8 1]
      swap A[2] with A[4] -> A = [16 14 10 4 7 9 3 2 8 1]
      swap A[4] with A[9] -> A = [16 14 10 8 7 9 3 2 4 1]
      after this call: A = [16 14 10 8 7 9 3 2 4 1]

result: A = [16 14 10 8 7 9 3 2 4 1]
exchanges performed: 7

         n     exchanges   exchanges/n   (n lg n)/n = lg n
      1000           742         0.742             9.000
     10000          7469         0.747            13.000
    100000         74242         0.742            16.000
   1000000        743123         0.743            19.000
  10000000       7440513         0.744            23.000
```

Every line of the hand trace in Section 5.5.3 is reproduced exactly, down to the intermediate array after each swap.

Now look at the table at the bottom, because it is the most convincing piece of evidence in this chapter. As n grows by a factor of ten thousand — from one thousand to ten million — the

quantity $\lg n$ climbs from 9 to 23, roughly two and a half times. If `BUILD-MAX-HEAP` were $\Theta(n \lg n)$, the exchanges-per-element figure would have to climb by a similar factor. **It does not move.** It sits at 0.74, then 0.75, then 0.74, then 0.74, then 0.74. Exchanges per element is a constant. That constant times n is $\Theta(n)$, exactly as the summation predicted.

> **A SANITY CHECK YOU CAN DO YOURSELF.** Why about 0.74 exchanges per element, rather than 1 or 2? Because more than half the nodes are leaves and are never even visited, and of the internal nodes that are visited, most sink only a level or two. Exercise 5.20 estimates the constant from the heights of the nodes and lands near $\frac{3}{4}$, which is just what the measurement shows.

5.6 Heapsort

5.6.1 The algorithm

Heapsort was published by J. W. J. Williams in 1964 [1], and Robert Floyd gave the linear-time construction of Section 5.5 in the same year [2]. We now have everything. Recall the plan from Section 5.2: run Selection Sort, but make "find the maximum" cheap by keeping the data in a heap. **Pseudocode 5.3** is the result.

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)
```

How it works, in words. After line 1, $A[1]$ holds the largest element of the whole array. In a sorted array the largest element belongs in the very last position, $A[n]$ — so line 3 puts it there, exchanging it with whatever was sitting in $A[n]$. That element is now in its final, correct, permanent place; it will never be touched again. Line 4 records this by shrinking the heap so that position n is no longer part of it. Line 5 repairs the one violation that the exchange introduced — the root now holds a small value from the bottom of the heap — and after the repair $A[1]$ holds the second-largest element. Repeat with $i = n - 1$, then $n - 2$, and so on.

The array therefore splits into two regions that move past each other:

- $A[1 \ldots \text{heap-size}]$ — still a max-heap, containing the elements not yet placed;
- $A[\text{heap-size} + 1 \ldots n]$ — already sorted, containing the largest elements in their final positions.

Each iteration moves the boundary one step to the left.

Note the economy of the exchange on line 3. Selection Sort had to *find* the maximum before it could place it; Heapsort already knows where it is. And the element it displaces does not need a home — it goes to the root, which is precisely where `MAX-HEAPIFY` expects a misplaced value to be.

5.6.2 A complete trace

We continue with the array from Section 5.5, so that line 1 produces the heap we have already built by hand: $A = [16,14,10,8,7,9,3,2,4,1]$, drawn in **Figure 5.13** as panel (a).

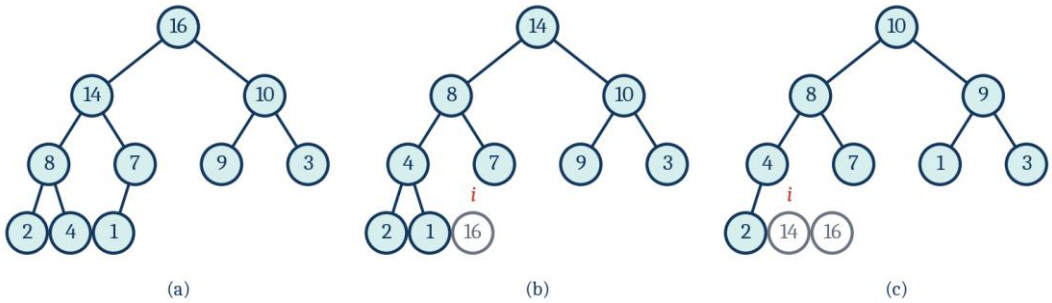


Figure 5.13 Heapsort, steps (a)–(c). Shaded nodes are still in the heap; unshaded ones have been placed in their final sorted positions and are no longer part of it. **(a)** the heap after `BUILD-MAX-HEAP`. **(b)** after exchanging 16 into position 10 and repairing the root. **(c)** after exchanging 14 into position 9 and repairing.

Iteration $i = 10$.

- Exchange $A[1] = 16$ with $A[10] = 1$. The array becomes [1,14,10,8,7,9,3,2,4 | 16], where the bar marks the sorted tail.
- Shrink: heap-size = 9. The 16 is now outside the heap, permanently.
- `MAX-HEAPIFY(A, 1)`: root 1, children 14 and 10; largest is 14 at index 2; swap. Now at index 2: value 1, children $A[4] = 8$ and $A[5] = 7$; largest is 8; swap. Now at index 4: value 1, children $A[8] = 2$ and $A[9] = 4$; largest is 4 at index 9; swap. Now at index 9: $2 \times 9 = 18 > 9$, a leaf. Stop.
- Result: [14,8,10,4,7,9,3,2,1 | 16] — panel (b).

Iteration $i = 9$.

- Exchange $A[1] = 14$ with $A[9] = 1$: [1,8,10,4,7,9,3,2 | 14,16].
- Shrink to heap-size = 8.
- `MAX-HEAPIFY(A, 1)`: root 1 against 8 and 10; swap with 10 at index 3. At index 3: value 1, children $A[6] = 9$ and $A[7] = 3$; swap with 9 at index 6. At index 6: $12 > 8$, a leaf. Stop.
- Result: [10,8,9,4,7,1,3,2 | 14,16] — panel (c).

The next three iterations are shown in **Figure 5.14**.

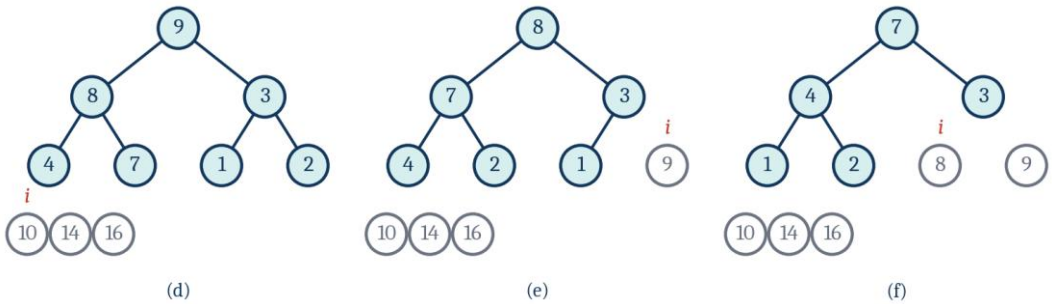


Figure 5.14 Heapsort, steps (d)–(f). The heap shrinks by one node each time and the sorted tail, drawn unshaded and laid out along the bottom, grows to meet it. **(d)** 10 placed. **(e)** 9 placed. **(f)** 8 placed.

Iteration $i = 8$. Exchange 10 with $A[8] = 2$; shrink to 7; heapify. Result [9,8,3,4,7,1,2 | 10,14,16] — panel (d).

Iteration $i = 7$. Exchange 9 with $A[7] = 2$; shrink to 6; heapify. Result [8,7,3,4,2,1 | 9,10,14,16] — panel (e).

Iteration $i = 6$. Exchange 8 with $A[6] = 1$; shrink to 5; heapify. Result [7,4,3,1,2 | 8,9,10,14,16] — panel (f).

Three more, in **Figure 5.15**.

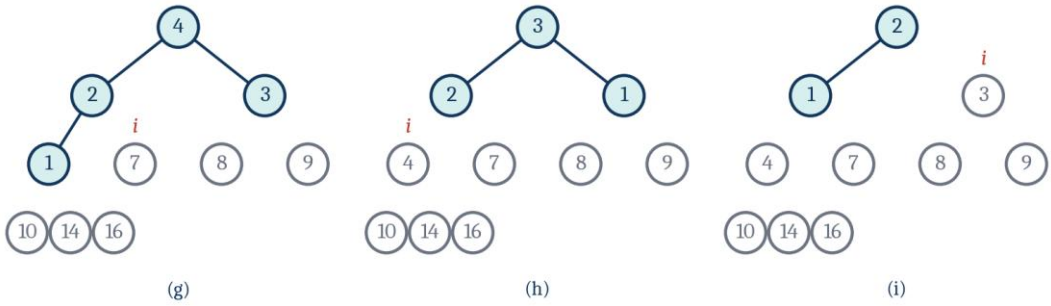


Figure 5.15 Heapsort, steps (g)–(i). With the heap down to four, three and two nodes the work per iteration has become trivial: the height is 2, then 1, then 1.

Iteration $i = 5$**.** Exchange 7 with $A[5] = 2$; shrink to 4; heapify. Result [4,2,3,1 | 7,8,9,10,14,16] — panel (g).

Iteration $i = 4$**.** Exchange 4 with $A[4] = 1$; shrink to 3; heapify. Result [3,2,1 | 4,7,8,9,10,14,16] — panel (h).

Iteration $i = 3$**.** Exchange 3 with $A[3] = 1$; shrink to 2; heapify. Result [2,1 | 3,4,7,8,9,10,14,16] — panel (i).

And the last one, in **Figure 5.16**.

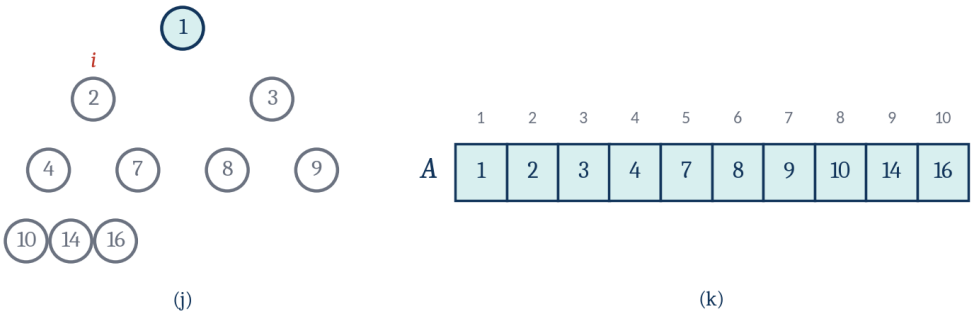


Figure 5.16 Heapsort, steps (j) and (k). (j) after the last exchange, the heap holds a single element. **(k)** the sorted array $A = [1,2,3,4,7,8,9,10,14,16]$, in non-decreasing order.

Iteration $i = 2$**.** Exchange 2 with $A[2] = 1$; shrink to 1; the call to `MAX-HEAPIFY(A, 1)` finds a one-element heap and returns at once. Result [1 | 2,3,4,7,8,9,10,14,16] — panel (j).

The loop stops at $i = 2$; there is no iteration for $i = 1$, and there does not need to be, because a heap of one element is a sorted array of one element. The final array is

$$A = [1,\ 2,\ 3,\ 4,\ 7,\ 8,\ 9,\ 10,\ 14,\ 16]$$

— panel (k). ✓

5.6.3 Correctness, by loop invariant

> **LOOP INVARIANT.** At the start of each iteration of the `for` loop of line 2, the subarray $A[1 \dots i]$ is a max-heap containing the i smallest elements of the original array, and the subarray $A[i + 1 \dots n]$ contains the $n - i$ largest elements of the original array, in sorted order.

Initialization. Before the first iteration $i = n$. Line 1 has just made $A[1 \dots n]$ a max-heap, and it trivially contains the n smallest elements of the array — all of them. The subarray $A[n + 1 \dots n]$ is empty, and an empty array is sorted. ✓

Maintenance. Assume the invariant at the start of an iteration with counter i. Since $A[1 \dots i]$ is a max-heap, $A[1]$ is the largest of those i elements — and since those are the i smallest of the original array, $A[1]$ is the $(n-i+1)$-th largest element overall, whose correct sorted position is exactly index i. Line 3 puts it there. Line 4 removes position i from the heap. The tail $A[i \dots n]$ now holds the $n-i+1$ largest elements in sorted order, because the element just placed is no larger than every element already in $A[i+1 \dots n]$ and no smaller than everything left in the heap. Meanwhile $A[1 \dots i-1]$ contains the $i-1$ smallest elements, and it satisfies the heap property everywhere except possibly at the root, where the exchange deposited a former leaf; the children of the root are unchanged and are still heap roots, so line 5's `MAX-HEAPIFY(A, 1)` restores the heap property. The invariant holds with i replaced by $i-1$. ✓

Termination. The loop ends with $i = 1$. The invariant says $A[2 \dots n]$ holds the $n-1$ largest elements in sorted order, and $A[1 \dots 1]$ holds the single smallest element. So $A[1 \dots n]$ is sorted in non-decreasing order. ✓ ■

5.6.4 The running time

Table 5.3 accounts for every line.

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			$\mathbf{O(n\lg n)}$

Adding up: $O(n) + \Theta(n) + O(n\lg n) = O(n\lg n)$, because $n\lg n$ dominates n.

Is the bound tight — is Heapsort $\Theta(n\lg n)$ rather than merely $O(n\lg n)$? For the **worst case** the cleanest proof is one we reach at the end of the chapter: Section 5.11 shows that *every* comparison sort needs $\Omega(n\lg n)$ comparisons on some input, and Heapsort is a comparison sort. So its worst case is exactly $\Theta(n\lg n)$.

The **best case** needs more care. Consider the first $n/2$ iterations of the loop. During each of them the heap still has at least $n/2$ elements, hence height at least $\lg(n/2) = \lg n - 1$. The element swapped to the root came from the very bottom of the heap, so it is one of the smaller elements present, and it usually sinks most of the way back down. If each of those $n/2$ iterations costs $\Omega(\lg n)$, the total is $\Omega\left(\frac{n}{2}(\lg n - 1)\right) = \Omega(n\lg n)$. Turning "usually" into a proof is hard, but it can be done: when all the keys are **distinct**, Heapsort makes $\Omega(n\lg n)$ comparisons on *every* input (a starred exercise in CLRS [3]). So, for distinct keys,

$$T_{\text{Heapsort}}(n) = \Theta(n\lg n) \quad \text{in the worst case, the best case, and every case in between.}$$

There is one honest exception. If all n keys are **equal**, every call to `MAX-HEAPIFY` finds that the root already dominates its children (the test is a strict $>$) and returns after $\Theta(1)$ work, so the whole sort takes only $\Theta(n)$ time.

That uniformity is Heapsort's distinguishing virtue. Apart from that degenerate all-equal input, there is no input on which it is slow, and there is no input on which it is fast.

HEAPSORT, SUMMARIZED. Worst case $\Theta(n\lg n)$. Best case $\Theta(n\lg n)$ when the keys are distinct. Average case $\Theta(n\lg n)$. **In-place**: it uses $O(1)$ extra space — a handful of index variables and one temporary for the swap. **Not stable.** It achieves the dream stated in Section 5.1.2.

5.6.5 Why Heapsort is not stable

A sort is **stable** if elements with equal keys keep their original relative order. Heapsort is not, and it is easy to see why: the very first thing it does is fling $A[1]$ to position n across the whole array, with no regard at all for what lies between. Two equal keys can be reordered by a single such exchange.

A TWO-ELEMENT COUNTEREXAMPLE. Sort the pairs $\langle 1, a\rangle, \langle 1, b\rangle$ by their first component. `BUILD-MAX-HEAP` leaves the array alone, because $1 \geq 1$. Then $i = 2$: exchange $A[1] \leftrightarrow A[2]$, producing $\langle 1, b\rangle, \langle 1, a\rangle$. The b has overtaken the a. Stability is lost on an array of length two.

We come back to stability, and to why it matters in real software, in Section 5.12.2.

5.6.6 Heapsort in C++

Listing 5.4 is a complete Heapsort that prints the array after every extraction, with a bar showing the boundary between the shrinking heap and the growing sorted tail.

Listing 5.4 A complete, runnable Heapsort that narrates every extraction.

```
// 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 5.4 (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;
}
```

Its output:

```
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
```

Compare line by line with panels (b) through (j) of **Figure 5.13**, **Figure 5.14**, **Figure 5.15** and **Figure 5.16**. Every intermediate array matches the figures exactly. That is what a correct implementation looks like: not merely a sorted answer at the end, but the same intermediate states the theory predicts.

5.7 Priority Queues

5.7.1 What a priority queue is

Heapsort is the heap's most famous *exercise*. Priority queues are the heap's most famous *use*, and in practice they matter more.

> **DEFINITION.** A **priority queue** is an abstract data type maintaining a changing set S of elements, each of which carries a **key** that represents its priority. A **max-priority queue** supports four operations: Insert(S, x) — add the element x to the set; Maximum(S) — return the element with the largest key, without removing it; Extract − Max(S) — remove and return the element with the largest key; Increase − Key(S, x, k) — raise the key of element x to the new value k, which must be at least as large as the current key. A **min-priority queue** is the mirror image, with Minimum, Extract − Min and Decrease − Key.

The phrase "abstract data type" means we are specifying *what* the operations do, not *how*. That separation matters: the same four operations can be implemented in several ways, with very different costs, and the heap is simply the best general answer.

Contrast it with an ordinary queue. A FIFO queue serves whoever arrived first; a priority queue serves whoever is most important, regardless of arrival order. If three customers walk into a laundry — one with an ordinary wash due tomorrow, one with an express order due in two hours, one with a wash due this afternoon — a FIFO queue serves them in the order they came through the door. A priority queue serves the express order first. In a computer system, the "customers" are jobs, packets, events or web pages awaiting a scan.

5.7.2 Where priority queues are used

- **Operating system job scheduling.** Processes carry priorities; the scheduler runs the highest-priority runnable process. When a process finishes or blocks, the scheduler performs an `EXTRACT-MAX`. When a new process is created it performs an `INSERT`. When a process has waited too long and must be boosted to avoid starvation, that is an `INCREASE-KEY`.
- **Discrete-event simulation.** The queue holds pending events keyed by the time they are due to happen, and the simulator repeatedly extracts the *earliest* — so this one is a min-priority queue. Simulating a factory, a network, or a queue of customers all work this way.

- **Graph algorithms.** Dijkstra's shortest-path algorithm (Chapter 13) keeps the unvisited vertices in a min-priority queue keyed by their current tentative distance; each relaxation of an edge is a `DECREASE-KEY`. Prim's minimum spanning tree algorithm (Chapter 11) does the same with edge weights. A* search adds a heuristic to the key and is otherwise identical.
- **Huffman coding.** Building an optimal prefix code (Chapter 9) means repeatedly extracting the two least-frequent symbols and inserting their merged node — two `EXTRACT-MIN`s and one `INSERT` per step.
- **Crawl and scan scheduling.** Which page do we fetch next? The one whose urgency score is highest. We return to this in Section 5.13.

5.7.3 `MAXIMUM`: the easy one

Pseudocode 5.4 needs only one line of real work.

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]
```

One array read. $O(1)$. No comparisons at all. This is the whole reason we chose a heap: the answer to "what is the biggest?" is permanently parked in slot 1 by the heap property, as proved in Section 5.3.5.

5.7.4 `EXTRACT-MAX`

Removing the maximum is the interesting operation, and **Pseudocode 5.5** gives its pseudocode.

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
```

The trick is the same one Heapsort uses on line 3. We cannot simply delete $A[1]$ and leave a hole — the array would no longer be a nearly complete tree. So we take the element from the *last* position, which is the only position whose removal keeps the shape legal, and drop it into the root. The shape is then correct, the heap property is broken in exactly one place, and `MAX-HEAPIFY` is the tool for exactly that.

> **WORKED EXAMPLE.** Extract from $A = [16,14,10,8,7,9,3,2,4,1]$, heap-size $= 10$. Step 1: save max $= A[1] = 16$. Step 2: $A[1] \leftarrow A[10] = 1$. The array is now $[1,14,10,8,7,9,3,2,4,1]$. Step 3: heap-size $\leftarrow 9$, so the trailing copy of 1 is outside the heap and irrelevant. Step 4: `MAX-HEAPIFY(A, 1)`. Root 1 against children 14 and 10: swap with 14. At index 2: value 1 against 8 and 7: swap with 8. At index 4: value 1 against $A[8] = 2$ and $A[9] = 4$: swap with 4. At index 9: leaf, stop. Step 5: return 16. The heap is $[14,8,10,4,7,9,3,2,1]$. Cost: one root-to-leaf walk, so $O(\lg n)$.

5.7.5 `INCREASE-KEY`

Raising an element's priority is the mirror image of `MAX-HEAPIFY`: instead of a value being too small and sinking, a value is now too large and must **float up**. **Pseudocode 5.6** lets it rise.

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)
```

The `while` loop compares the element only with its parent, and stops the moment the parent wins. There is no need to look at siblings: if the new value is smaller than its parent, the heap property holds locally, and everything above the parent was already correct.

> **WORKED EXAMPLE.** Take the heap [10,8,9,4,7,3,2,1] with heap-size = 8, and raise the key at position 8 from 1 to 20. Step 1: $20 \geq 1$, so the call is legal. Set $A[8] \leftarrow 20$. Step 2: $i = 8$. Parent(8) = 4. Is $A[4] = 4 < 20$? Yes — swap. Now $i = 4$. Step 3: Parent(4) = 2. Is $A[2] = 8 < 20$? Yes — swap. Now $i = 2$. Step 4: Parent(2) = 1. Is $A[1] = 10 < 20$? Yes — swap. Now $i = 1$. Step 5: $i = 1$, so the loop condition $i > 1$ fails. Stop. Result: [20,10,9,8,7,3,2,4]. The 20 rose three levels; everything it passed moved down one. Cost: at most one leaf-to-root walk, so $O(\lg n)$.

5.7.6 `INSERT`

And now the pretty part. Insertion needs no new code at all: it is `INCREASE-KEY` in disguise. **Pseudocode 5.7** shows it in pseudocode.

Pseudocode 5.7 `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
```

The idea: grow the heap by one position at the end — the only position where growth keeps the shape legal — put a value of $-\infty$ there, which cannot possibly violate the heap property because nothing is smaller than $-\infty$, and then call `INCREASE-KEY` to raise it to the real key. The float-up loop then carries it to wherever it belongs. **Pseudocode 5.8** writes this out.

Pseudocode 5.8 `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)
```

Note the small optimization in **Pseudocode 5.7**: rather than exchanging two values at every step, it *shifts the parent down* into the child's slot and only writes the key once, at the end. Same result, half the memory writes.

5.7.7 What the four operations cost

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`	$O(\lg n)$	$O(1)$	$O(n)$

Table 5.4 is the argument for heaps in a single picture. An unsorted array makes insertion free but extraction linear. A sorted array makes extraction free but insertion linear. Either is fine if you do almost all of one and almost none of the other — but a real priority queue does a mixture, and then the linear operation dominates everything. The heap has no cheap operation and no expensive one; it has four logarithmic-or-better operations, and $\lg n$ is so nearly constant in practice that this is the right trade.

> **THE ARITHMETIC OF THE TRADE.** Suppose a scheduler performs one million insertions and one million extractions on a queue holding about one million jobs. With an unsorted array: $10^6 O(1) + 10^6 O(10^6) = $ about 10^{12} operations — around seventeen minutes at a billion operations a second. With a heap: $2 ^6 O(20) = $ about 4×10^7 operations — around four hundredths of a second.

5.7.8 A complete priority queue in C++

Listing 5.5 implements all four operations as a class, and exercises every one of them.

Listing 5.5 A complete max-priority queue, with all four operations exercised.

```
// A max-priority queue on a binary heap: MAXIMUM, EXTRACT-MAX, INCREASE-KEY, INSERT.
#include <iostream>
#include <vector>
#include <string>
#include <climits>
#include <stdexcept>
using namespace std;

class MaxPriorityQueue {
    vector<int> A;                              // A[0] unused; the heap lives in
A[1..heapSize]
    int heapSize = 0;

    void maxHeapify(int i) {                    // sink A[i] to its rightful depth
        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); }
    }

public:
    MaxPriorityQueue() : A(1, 0) {}             // A[0] is a placeholder, never read

    int size() const { return heapSize; }

    string toString() const {
        string s = "[";
        for (int i = 1; i <= heapSize; ++i) { s += to_string(A[i]); if (i < heapSize) s +=
" "; }
        return s + "]";
    }

    int maximum() const {                       // O(1): the answer is simply the root
        if (heapSize < 1) throw runtime_error("heap underflow");
        return A[1];
    }

    int extractMax() {                          // O(lg n)
        if (heapSize < 1) throw runtime_error("heap underflow");
        int best = A[1];
        A[1] = A[heapSize];                     // the last leaf moves to the root ...
        --heapSize;                             // ... the heap shrinks ...
        maxHeapify(1);                          // ... and the new root sinks into place
        return best;
    }
```

Listing 5.5 (Continued) *A complete max-priority queue, with all four operations exercised.*

```
    void increaseKey(int i, int key, bool verbose = false) {   // O(lg n): float A[i]
upward
        if (i < 1 || i > heapSize) throw runtime_error("index outside the heap");
        if (key < A[i]) throw runtime_error("new key is smaller than current key");
        A[i] = key;
        while (i > 1 && A[i / 2] < A[i]) {       // while the parent is beaten by the child
            if (verbose) cout << "      parent A[" << i / 2 << "]=" << A[i / 2]
                              << " < A[" << i << "]=" << A[i] << ", so swap them\n";
            swap(A[i], A[i / 2]);
            i = i / 2;
        }
        if (verbose) cout << "      the key has stopped at position " << i << "\n";
    }

    void insert(int key) {                       // O(lg n): grow by one, then float up
        ++heapSize;
        if ((int)A.size() <= heapSize) A.push_back(INT_MIN); else A[heapSize] = INT_MIN;
        increaseKey(heapSize, key);              // -infinity can only rise
    }
};

int main() {
    MaxPriorityQueue q;
    cout << "INSERT in the order 4, 1, 3, 2, 16, 9, 10, 14, 8, 7\n";
    for (int k : {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}) {
        q.insert(k);
        cout << "  insert " << k << "\t-> " << q.toString() << "\n";
    }
    cout << "\nMAXIMUM = " << q.maximum() << "  (one array read, no comparisons)\n";

    cout << "\nINSERT(15) into the heap above\n";
    q.insert(15);
    cout << "  -> " << q.toString() << "\n";

    cout << "\nEXTRACT-MAX three times\n";
    for (int t = 0; t < 3; ++t)
        cout << "  extracted " << q.extractMax() << "\t-> " << q.toString() << "\n";

    cout << "\nINCREASE-KEY(A, 8, 20): raise the key at position 8 to 20\n";
    q.increaseKey(8, 20, true);
    cout << "  -> " << q.toString() << "\n";
```

Listing 5.5 (Continued) *A complete max-priority queue, with all four operations exercised.*

```
    cout << "\nDrain the queue: ";
    while (q.size() > 0) cout << q.extractMax() << " ";
    cout << "\n";
    return 0;
}
```

The output:

```
INSERT in the order 4, 1, 3, 2, 16, 9, 10, 14, 8, 7
  insert 4  -> [4]
  insert 1  -> [4 1]
  insert 3  -> [4 1 3]
  insert 2  -> [4 2 3 1]
  insert 16 -> [16 4 3 1 2]
  insert 9  -> [16 4 9 1 2 3]
  insert 10 -> [16 4 10 1 2 3 9]
  insert 14 -> [16 14 10 4 2 3 9 1]
  insert 8  -> [16 14 10 8 2 3 9 1 4]
  insert 7  -> [16 14 10 8 7 3 9 1 4 2]

MAXIMUM = 16  (one array read, no comparisons)

INSERT(15) into the heap above
  -> [16 15 10 8 14 3 9 1 4 2 7]

EXTRACT-MAX three times
  extracted 16  -> [15 14 10 8 7 3 9 1 4 2]
  extracted 15  -> [14 8 10 4 7 3 9 1 2]
  extracted 14  -> [10 8 9 4 7 3 2 1]

INCREASE-KEY(A, 8, 20): raise the key at position 8 to 20
      parent A[4]=4 < A[8]=20, so swap them
      parent A[2]=8 < A[4]=20, so swap them
      parent A[1]=10 < A[2]=20, so swap them
      the key has stopped at position 1
  -> [20 10 9 8 7 3 2 4]

Drain the queue: 20 10 9 8 7 4 3 2
```

Two things in that output repay attention.

First, look at the array after inserting the ten numbers: [16,14,10,8,7,3,9,1,4,2]. Compare it with the array `BUILD-MAX-HEAP` produced from the *same ten numbers* in Section 5.5.3: [16,14,10,8,7,9,3,2,4,1]. **They are different heaps.** Both are valid — check the parent-child inequalities in each and you will find no violation — which is the point made back in Section 5.3.4: a set of numbers has many legal heaps. Note also the cost difference. Ten `INSERT`s cost $O(n\lg n)$; one `BUILD-MAX-HEAP` costs $O(n)$. If you have all the data up front, build; if it arrives one item at a time, insert.

Second, the `INCREASE-KEY` trace reproduces the hand trace of Section 5.7.5 exactly: three swaps, three levels, finishing at the root.

5.7.9 Using the standard library instead

In production — and in a programming contest, where time is scored [4] — you would not write the class above. C++ ships with it, and the standard guarantees the complexities [5]. **Listing 5.6** shows the four ways you will actually use heaps in C++.

Listing 5.6 Heaps from the C++ standard library: `std::priority_queue`, custom comparators, and the raw `make_heap` / `sort_heap` algorithms.

```
// The same jobs done with the standard library: std::priority_queue and the heap
algorithms.
#include <iostream>
#include <queue>
#include <vector>
#include <string>
#include <algorithm>
#include <functional>
using namespace std;

struct ScanJob {                                    // a crawl job waiting for its turn
    int  urgency;                                   // bigger means more urgent
    string url;
};
struct ByUrgency {                                  // "less urgent" comes first in the
ordering,
    bool operator()(const ScanJob& a, const ScanJob& b) const {
        return a.urgency < b.urgency;               // so the TOP of the queue is the most
urgent
    }
};

int main() {
    // (1) A max-heap of plain integers: this is the default.
    priority_queue<int> maxQ;
    for (int k : {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}) maxQ.push(k);
    cout << "max-heap drain: ";
    while (!maxQ.empty()) { cout << maxQ.top() << " "; maxQ.pop(); }
    cout << "\n";

    // (2) A min-heap: same container, reversed comparator.
    priority_queue<int, vector<int>, greater<int>> minQ;
    for (int k : {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}) minQ.push(k);
    cout << "min-heap drain: ";
    while (!minQ.empty()) { cout << minQ.top() << " "; minQ.pop(); }
    cout << "\n";

    // (3) A heap of records, ordered by one field.
    priority_queue<ScanJob, vector<ScanJob>, ByUrgency> jobs;
    jobs.push({3, "/pricing"});
    jobs.push({9, "/"});
    jobs.push({5, "/blog/geo-basics"});
    jobs.push({7, "/products"});
    cout << "scan order: ";
    while (!jobs.empty()) { cout << jobs.top().url << "(" << jobs.top().urgency << ") ";
jobs.pop(); }
    cout << "\n";
```

Listing 5.6 (Continued) *Heaps from the C++ standard library: std::priority_queue, custom comparators, and the raw make_heap / sort_heap algorithms.*

```
    // (4) The raw heap algorithms on a vector you own. Note make_heap is the O(n) build.
    vector<int> v = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7};
    make_heap(v.begin(), v.end());                  // BUILD-MAX-HEAP, in O(n)
    cout << "make_heap:      ";
    for (int x : v) cout << x << " ";
    cout << "\n";
    sort_heap(v.begin(), v.end());                  // the extraction loop of HEAPSORT
    cout << "sort_heap:      ";
    for (int x : v) cout << x << " ";
    cout << "\n";
    return 0;
}
```

Its output:

```
max-heap drain: 16 14 10 9 8 7 4 3 2 1
min-heap drain: 1 2 3 4 7 8 9 10 14 16
scan order: /(9) /products(7) /blog/geo-basics(5) /pricing(3)
make_heap:      16 14 10 8 7 9 3 2 4 1
sort_heap:      1 2 3 4 7 8 9 10 14 16
```

Look at the fourth line. `make_heap` produced [16,14,10,8,7,9,3,2,4,1] — **character for character the array our own `BUILD-MAX-HEAP` produced** in Section 5.5.7, and the array in **Figure 5.12**. The standard library implements exactly the algorithm of this chapter.

> **DIRECTION TRAP.** `std::priority_queue<int>` is a **max**-heap: `top()` gives the largest. Python's `heapq` functions build a **min**-heap by default — Python 3.14 added separate `heapify_max`, `heappush_max` and `heappop_max` functions for max-heaps [6] — and Java's `PriorityQueue` is a min-heap too. Move between languages carelessly and your Dijkstra will find the longest path. To flip a C++ queue to a min-heap, pass `greater<int>` as the third template argument, as in case (2) above — or negate every key on the way in and again on the way out, which works but is easy to get wrong with unsigned or floating-point keys.

5.8 Quicksort

5.8.1 The idea

Quicksort was invented by Tony Hoare in 1959 and published in 1962 [7], while he was a visiting student in Moscow working on a machine-translation project that needed to sort Russian words before looking them up in a dictionary. Sixty-five years later it is still the sort that most standard libraries reach for first. It has a $\Theta(n^2)$ worst case, which sounds disqualifying — and yet it beats Heapsort, which has no bad case at all, on almost every real input. Understanding *why* is the most instructive thing in this chapter.

> **DEFINITION. Quicksort** is a divide-and-conquer sorting algorithm. Given a subarray $A[p \dots r]$, it chooses an element called the **pivot**, rearranges the subarray so that every element no greater than the pivot comes before every element no smaller than it, and then sorts the two resulting pieces recursively. There is no combine step: once both pieces are sorted, the whole subarray is sorted, in place.

5.8.2 Divide and conquer, with the work moved

Chapter 3 gave us the three-step divide-and-conquer template. Quicksort fills it in like this:

1. **Divide.** Partition $A[p \dots r]$ into two subarrays $A[p \dots q-1]$ and $A[q+1 \dots r]$ such that every element of the first is $\leq A[q]$ and every element of the second is $\geq A[q]$. Computing the index q and doing the rearrangement is the whole of the work.
2. **Conquer.** Sort the two subarrays by recursive calls to Quicksort.
3. **Combine.** Nothing. The subarrays are sorted in place, and the pivot already sits between them, so $A[p \dots r]$ is sorted the moment the two recursive calls return.

Compare with Merge Sort from Chapter 3:

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 names the real trade. Merge Sort *buys* its guaranteed even split by paying for a merge buffer. Quicksort refuses to pay, and in exchange accepts that the split is whatever the data happens to give it. Sections 5.9 and 5.10 are about how bad that can get and how to stop it getting bad.

5.8.3 Lomuto's `PARTITION`, and the four-region invariant

We present two partition schemes. The first, Lomuto's, is the one in the current edition of CLRS [3] and the one to write if you are writing your first Quicksort: it has one loop, one direction, and an invariant you can see. **Pseudocode 5.9** gives it in full.

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
```

The loop variable j sweeps left to right, once. At every moment the subarray is divided into **four regions**:

```
p                 i        j                 r
|------------------|--------|-----------------|---|
|     ≤ x          |  > x   |    unexamined   | x |
|------------------|--------|-----------------|---|
A[p..i]            A[i+1..j-1]  A[j..r-1]     A[r]
```

> **LOOP INVARIANT (four regions).** At the start of each iteration of the `for` loop of line 3, for any array index k: 1. if $p \le k \le i$ then $A[k] \le x$; 2. if $i + 1 \le k \le j - 1$ then $A[k] > x$; 3. if $k = r$ then $A[k] = x$. Indices j through $r - 1$ are the unexamined region, about which nothing is claimed.

Initialization. Before the first iteration, $i = p - 1$ and $j = p$. Region 1 is $A[p \dots p - 1]$ — empty, so condition 1 holds vacuously. Region 2 is $A[p \dots p - 1]$ — also empty, so condition 2 holds vacuously. Condition 3 holds because line 1 set $x = A[r]$ and nothing has been moved yet. ✓

Maintenance. Consider an iteration with the invariant holding. Two cases:

- $A[j] > x$**.** Lines 5–6 are skipped and only j increments. The element at index j, which is $>x$, now falls inside region 2, whose defining property is exactly "$>x$". Condition 2 still holds. Conditions 1 and 3 are untouched. ✓

- $A[j] \le x$**.** Line 5 increments i, so index i now points at the *first* element of the old region 2 — a value $>x$. Line 6 swaps that value with $A[j]$, which is $\le x$. Afterward: $A[i] \le x$, so it correctly joins region 1; and the value $>x$ that was at $A[i]$ has moved to index j, which becomes the last slot of region 2 when j increments. Both conditions 1 and 2 hold again. ✓

Termination. The loop ends when $j = r$. The unexamined region $A[j \dots r-1] = A[r \dots r-1]$ is empty, so every element of $A[p \dots r-1]$ lies in region 1 or region 2, that is, is either $\le x$ or $>x$. Line 7 then exchanges $A[i+1]$ — the first element of region 2, so a value $>x$ — with the pivot at $A[r]$. The pivot lands at index $i+1$, with everything $\le x$ to its left and everything $>x$ to its right. Line 8 returns $q = i+1$, and the postcondition Equation (5.9)

$$A[k] \le A[q] \text{ for } p \le k < q, \qquad A[k] > A[q] \text{ for } q < k \le r \tag{5.9}$$

is established. ✓ ■

Running time. The `for` loop runs $r - p$ times, and each iteration does $O(1)$ work. With $n = r - p + 1$, that is $\Theta(n)$.

5.8.4 A Lomuto partition, traced by hand

Partition $A = [2,8,7,1,3,5,6,4]$, indices 1 through 8. The pivot is $x = A[8] = 4$. We start with $i = 0$.

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 $A[1] \leftrightarrow A[1]$	$[\underline{2}, 8,7,1,3,5,6,4]$
2	8	no	nothing	$[\underline{2}, 8,7,1,3,5,6,4]$
3	7	no	nothing	$[\underline{2}, 8,7,1,3,5,6,4]$
4	1	yes	$i \to 2$; swap $A[2] \leftrightarrow A[4]$	$[\underline{2,1}, 7,8,3,5,6,4]$
5	3	yes	$i \to 3$; swap $A[3] \leftrightarrow A[5]$	$[\underline{2,1,3}, 8,7,5,6,4]$
6	5	no	nothing	$[\underline{2,1,3}, 8,7,5,6,4]$
7	6	no	nothing	$[\underline{2,1,3}, 8,7,5,6,4]$

Table 5.6 carries the loop. After it, $i = 3$, so line 7 swaps $A[4] \leftrightarrow A[8]$, giving

$$A = [2,\ 1,\ 3,\ \mathbf{4},\ 7,\ 5,\ 6,\ 8],$$

and line 8 returns $q = 4$. Check the postcondition Equation (5.9): everything left of index 4 is $\{2,1,3\}$, all ≤ 4 ✓; everything right of it is $\{7,5,6,8\}$, all >4 ✓. The pivot 4 is now in its final sorted position and will never move again.

5.8.5 Hoare's `PARTITION`, the original scheme

The second scheme is Hoare's original one [7], in which two pointers start at opposite ends and walk toward each other. **Figure 5.17** shows it: the pseudocode on the left, and a full trace on the right. **Pseudocode 5.10** repeats the pseudocode as text, so that each line can carry a comment.

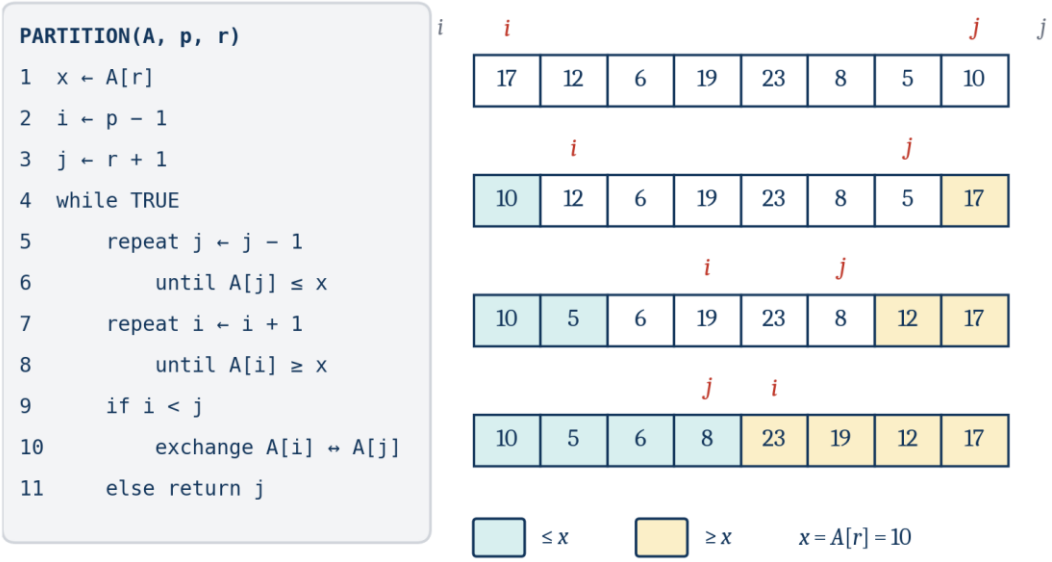


Figure 5.17 Hoare's PARTITION, with a full trace on [17,12,6,19,23,8,5,10]. Teal marks the elements known to be $\leq x$, amber those known to be $\geq x$. The pointers i and j begin just outside the array and close in; when they cross, the partition is finished and j is returned.

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
```

In words. Pointer j walks leftwards looking for something that belongs on the left (a value $\leq x$). Pointer i walks rightwards looking for something that belongs on the right (a value $\geq x$). When both have found a misplaced element, exchange them — each goes to the side where it belongs. When the pointers meet or cross, everything has been examined and j marks the boundary.

The difference from Lomuto is worth stating precisely, because it catches people out:

- Lomuto returns the **final position of the pivot**, and the pivot is excluded from both recursive calls: `QUICKSORT(A, p, q-1)` and `QUICKSORT(A, q+1, r)`.
- Hoare returns a **boundary index**, not the pivot's position. The pivot may be anywhere. So the recursive calls must be `QUICKSORT(A, p, q)` and `QUICKSORT(A, q+1, r)` — note the q, not $q - 1$. Writing $q - 1$ here loses an element; writing $q + 1$ for the left call and q for the right causes infinite recursion.

Hoare's version does about three times fewer exchanges than Lomuto's on random data, and it behaves far better when the array has many equal keys — which is why real libraries use variants of it. Lomuto's version is easier to prove and easier to remember. Learn Lomuto first, then graduate.

> **THE PIVOT TRAP IN HOARE'S SCHEME.** Line 1 of **Pseudocode 5.10**, like **Figure 5.17**, takes the pivot value from the **last** slot, $x = A[r]$. That is perfectly correct for a single partition, and the first trace below uses it. But it must not be combined with the recursion `QUICKSORT(A, p, q)` and `QUICKSORT(A, q+1, r)`. Suppose $A[r]$ happens to be the strict maximum of $A[p \ldots r]$.

Pointer j stops at once at r, because $A[r] \leq x$. Pointer i climbs past every element, all of them smaller than x, and first stops at r as well. Now $i = j = r$, so the procedure returns $q = r$, and the left call `QUICKSORT(A, p, r)` receives the very same subarray: the recursion never ends. The cure is to take the pivot value from the **first** slot, $x = A[p]$, as CLRS [3] does in its treatment of Hoare's scheme. Then the returned q always satisfies $p \leq q < r$, both pieces are strictly shorter than $A[p \ldots r]$, and the recursion is safe. The second trace below uses exactly this choice.

5.8.6 Hoare's partition, traced twice

Take the array $A = [17,12,6,19,23,8,5,10]$, indices $1 \ldots 8$.

First trace: the pivot is the last element, $x = A[8] = 10$. This is what the pseudocode's line 1 actually says, and it is what **Figure 5.17** draws.

- **Scan 1.** j starts at 9 and steps down: $A[8] = 10 \leq 10$ ✓ — stop at $j = 8$. i starts at 0 and steps up: $A[1] = 17 \geq 10$ ✓ — stop at $i = 1$. Since $1 < 8$, swap $A[1] \leftrightarrow A[8]$: [10,12,6,19,23,8,5,17].
- **Scan 2.** j steps down from 8: $A[7] = 5 \leq 10$ ✓ — stop at $j = 7$. i steps up from 1: $A[2] = 12 \geq 10$ ✓ — stop at $i = 2$. Since $2 < 7$, swap: [10,5,6,19,23,8,12,17].
- **Scan 3.** j steps down from 7: $A[6] = 8 \leq 10$ ✓ — stop at $j = 6$. i steps up from 2: $A[3] = 6 \geq 10$? No. $A[4] = 19 \geq 10$ ✓ — stop at $i = 4$. Since $4 < 6$, swap: [10,5,6,8,23,19,12,17].
- **Scan 4.** j steps down from 6: $A[5] = 23 \leq 10$? No. $A[4] = 8 \leq 10$ ✓ — stop at $j = 4$. i steps up from 4: $A[5] = 23 \geq 10$ ✓ — stop at $i = 5$. Now $i = 5 \not< j = 4$: the pointers have crossed. **Return $j = 4$.**

Left part $A[1 \ldots 4] = [10,5,6,8]$, every element ≤ 10 ✓. Right part $A[5 \ldots 8] = [23,19,12,17]$, every element ≥ 10 ✓.

Second trace: the pivot value is 17. Now take the pivot from the first slot, $x = A[1] = 17$ — the safe choice for use inside Quicksort, as the warning above explains. It is also instructive to see that the same procedure works for any pivot value drawn from the array.

- **Scan 1.** j: $A[8] = 10 \leq 17$ ✓, $j = 8$. i: $A[1] = 17 \geq 17$ ✓, $i = 1$. Swap: [10,12,6,19,23,8,5,17].
- **Scan 2.** j: $A[7] = 5 \leq 17$ ✓, $j = 7$. i: $A[2] = 12 \geq 17$? No. $A[3] = 6$? No. $A[4] = 19 \geq 17$ ✓, $i = 4$. Swap: [10,12,6,5,23,8,19,17].
- **Scan 3.** j: $A[6] = 8 \leq 17$ ✓, $j = 6$. i: $A[5] = 23 \geq 17$ ✓, $i = 5$. Swap: [10,12,6,5,8,23,19,17].
- **Scan 4.** j: $A[5] = 8 \leq 17$ ✓, $j = 5$. i: $A[6] = 23 \geq 17$ ✓, $i = 6$. Crossed. **Return $j = 5$.**

Left part [10,12,6,5,8], all ≤ 17 ✓. Right part [23,19,17], all ≥ 17 ✓.

Listing 5.7 runs all three traces and prints them, so that you can check the hand work against the machine.

Listing 5.7 Lomuto's and Hoare's partition schemes, each narrating its own steps. Indices here are 0-based, so subtract one from every index in the hand traces above.

```
// Two partition schemes, each traced step by step: Lomuto (CLRS 4th ed.) and Hoare (1961).
#include <iostream>
#include <vector>
#include <string>
using namespace std;

void dump(const vector<int>& A, int p, int r, const string& note) {
    cout << "    [";
    for (int k = p; k <= r; ++k) cout << A[k] << (k < r ? " " : "");
    cout << "]  " << note << "\n";
}

// ---------- Lomuto: one forward scan, four regions ----------
int lomutoPartition(vector<int>& A, int p, int r) {
    int x = A[r];                                   // the pivot is the last element
    cout << "  pivot x = A[" << r << "] = " << x << "\n";
    int i = p - 1;                                  // A[p..i] holds the values <= x
    for (int j = p; j <= r - 1; ++j) {
        cout << "  j = " << j << ", A[j] = " << A[j];
        if (A[j] <= x) {
            ++i;
            swap(A[i], A[j]);
            cout << " <= " << x << ": grow the small region to i = " << i
                 << " and swap A[" << i << "] with A[" << j << "]\n";
        } else {
            cout << " > " << x << ": leave it in the large region, i stays " << i << "\n";
        }
        dump(A, p, r, "");
    }
    swap(A[i + 1], A[r]);                           // drop the pivot between the two regions
    cout << "  put the pivot in its place: swap A[" << i + 1 << "] with A[" << r << "]\n";
    dump(A, p, r, "");
    return i + 1;
}

// ---------- Hoare: two scans that walk toward each other ----------
int hoarePartition(vector<int>& A, int p, int r, int pivotValue) {
    int x = pivotValue;
    cout << "  pivot value x = " << x << "\n";
    int i = p - 1, j = r + 1;
    int scan = 0;
    while (true) {
        ++scan;
        do { --j; } while (A[j] > x);               // walk left until a value <= x
        do { ++i; } while (A[i] < x);               // walk right until a value >= x
        cout << "  scan " << scan << ": j stops at index " << j << " (A[j]=" << A[j]
```

Listing 5.7 (Continued) *Lomuto's and Hoare's partition schemes, each narrating its own steps. Indices here are 0-based, so subtract one from every index in the hand traces above.*

```
                << "), i stops at index " << i << " (A[i]=" << A[i] << ")";
        if (i < j) {
            cout << "; i < j so swap\n";
            swap(A[i], A[j]);
            dump(A, p, r, "");
        } else {
            cout << "; i >= j so stop and return j = " << j << "\n";
            return j;
        }
    }
}

int main() {
    {
        cout << "LOMUTO PARTITION on [2 8 7 1 3 5 6 4]\n";
        vector<int> A = {2, 8, 7, 1, 3, 5, 6, 4};
        int q = lomutoPartition(A, 0, 7);
        cout << "  returns q = " << q << " (0-based), pivot sits at A[" << q << "] = " <<
A[q] << "\n\n";
    }
    {
        cout << "HOARE PARTITION on [17 12 6 19 23 8 5 10], pivot = last element = 10\n";
        vector<int> A = {17, 12, 6, 19, 23, 8, 5, 10};
        int q = hoarePartition(A, 0, 7, A[7]);
        cout << "  final array:";
        dump(A, 0, 7, "");
        cout << "  left part A[0.." << q << "] = ";
        for (int k = 0; k <= q; ++k) cout << A[k] << " ";
        cout << "\n  right part A[" << q + 1 << "..7] = ";
        for (int k = q + 1; k <= 7; ++k) cout << A[k] << " ";
        cout << "\n\n";
    }
    {
        cout << "HOARE PARTITION on [17 12 6 19 23 8 5 10], pivot = first element = 17\n";
        vector<int> A = {17, 12, 6, 19, 23, 8, 5, 10};
        int q = hoarePartition(A, 0, 7, A[0]);
        cout << "  final array:";
        dump(A, 0, 7, "");
        cout << "  left part A[0.." << q << "] = ";
        for (int k = 0; k <= q; ++k) cout << A[k] << " ";
        cout << "\n  right part A[" << q + 1 << "..7] = ";
        for (int k = q + 1; k <= 7; ++k) cout << A[k] << " ";
        cout << "\n";
    }
    return 0;
}
```

The output — note that C++ arrays start at 0, so the returned $q = 3$ corresponds to the $q = 4$ of the 1-based hand trace, and $q = 4$ to the hand trace's $q = 5$:

```
LOMUTO PARTITION on [2 8 7 1 3 5 6 4]
  pivot x = A[7] = 4
  j = 0, A[j] = 2 <= 4: grow the small region to i = 0 and swap A[0] with A[0]
    [2 8 7 1 3 5 6 4]
  j = 1, A[j] = 8 > 4: leave it in the large region, i stays 0
    [2 8 7 1 3 5 6 4]
  j = 2, A[j] = 7 > 4: leave it in the large region, i stays 0
    [2 8 7 1 3 5 6 4]
  j = 3, A[j] = 1 <= 4: grow the small region to i = 1 and swap A[1] with A[3]
    [2 1 7 8 3 5 6 4]
  j = 4, A[j] = 3 <= 4: grow the small region to i = 2 and swap A[2] with A[4]
    [2 1 3 8 7 5 6 4]
  j = 5, A[j] = 5 > 4: leave it in the large region, i stays 2
    [2 1 3 8 7 5 6 4]
  j = 6, A[j] = 6 > 4: leave it in the large region, i stays 2
    [2 1 3 8 7 5 6 4]
  put the pivot in its place: swap A[3] with A[7]
    [2 1 3 4 7 5 6 8]
  returns q = 3 (0-based), pivot sits at A[3] = 4

HOARE PARTITION on [17 12 6 19 23 8 5 10], pivot = last element = 10
  pivot value x = 10
  scan 1: j stops at index 7 (A[j]=10), i stops at index 0 (A[i]=17); i < j so swap
    [10 12 6 19 23 8 5 17]
  scan 2: j stops at index 6 (A[j]=5), i stops at index 1 (A[i]=12); i < j so swap
    [10 5 6 19 23 8 12 17]
  scan 3: j stops at index 5 (A[j]=8), i stops at index 3 (A[i]=19); i < j so swap
    [10 5 6 8 23 19 12 17]
  scan 4: j stops at index 3 (A[j]=8), i stops at index 4 (A[i]=23); i >= j so stop and
return j = 3
  final array:    [10 5 6 8 23 19 12 17]
  left part A[0..3] = 10 5 6 8
  right part A[4..7] = 23 19 12 17

HOARE PARTITION on [17 12 6 19 23 8 5 10], pivot = first element = 17
  pivot value x = 17
  scan 1: j stops at index 7 (A[j]=10), i stops at index 0 (A[i]=17); i < j so swap
    [10 12 6 19 23 8 5 17]
  scan 2: j stops at index 6 (A[j]=5), i stops at index 3 (A[i]=19); i < j so swap
    [10 12 6 5 23 8 19 17]
  scan 3: j stops at index 5 (A[j]=8), i stops at index 4 (A[i]=23); i < j so swap
    [10 12 6 5 8 23 19 17]
  scan 4: j stops at index 4 (A[j]=8), i stops at index 5 (A[i]=23); i >= j so stop and
return j = 4
  final array:    [10 12 6 5 8 23 19 17]
  left part A[0..4] = 10 12 6 5 8
  right part A[5..7] = 23 19 17
```

Every step of both Hoare traces matches the hand work, and the first of them matches **Figure 5.17** exactly.

5.8.7 The Quicksort procedure itself

With `PARTITION` in hand, Quicksort is the four lines of **Pseudocode 5.11**.

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 is `QUICKSORT(A, 1, n)`.

The base case is the condition on line 1. If $p \geq r$ the subarray has zero or one element, which is already sorted, and the procedure returns immediately. That is the whole termination argument: each recursive call receives a strictly shorter subarray, because the pivot at index q belongs to neither, so the recursion must reach length ≤ 1 after finitely many steps.

> **THE OFF-BY-ONE THAT BREAKS EVERYTHING.** If you use Lomuto's partition but write `QUICKSORT(A, p, q)` on line 3, the pivot is included in the left recursive call. When the pivot happens to be the largest element in the subarray, partition returns $q = r$, the left call is `QUICKSORT(A, p, r)` — the *same* subarray — and the program recurses for ever until the stack runs out. The rule is simple: **Lomuto excludes the pivot from both calls; Hoare excludes it from neither.** Match the recursion to the partition you actually wrote.

5.8.8 Quicksort, traced completely

Listing 5.8 sorts [2,8,7,1,3,5,6,4] and prints the whole array after each partition, with brackets marking the subarray currently being worked on.

Listing 5.8 Quicksort with Lomuto partitioning, narrating the recursion. Indentation shows the depth of the call.

```
// 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 5.8 (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;
}
```

The output:

```
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
```

Read it as a tree. The root call partitions around 4, producing the left piece [2,1,3] and the right piece [7,5,6,8]. The left piece partitions around 3, then around 1. The right piece partitions around 8 — an unlucky choice, since 8 is the largest element there and the right sub-piece is empty — then around 6.

Notice something the trace makes vivid: **a pivot never moves again.** Once `PARTITION` returns, $A[q]$ holds a value that is $\geq$ everything to its left and $\leq$ everything to its right, and both recursive calls stay strictly on one side of it. Five partitions place five elements permanently; the other three fall out as one-element base cases. That is the invariant that makes Quicksort correct, and it needs no merge to enforce.

5.9 Analyzing Quicksort: Best, Worst, and Average

5.9.1 Everything depends on the split

`PARTITION` always costs $\Theta(n)$ on a subarray of n elements — one pass, one comparison per element. So the running time of Quicksort is entirely determined by *how the array gets divided*. Write the general recurrence: if the partition puts k elements on the left, then

$$T(n) = T(k) + T(n-k-1) + \Theta(n), \tag{5.10}$$

where the -1 accounts for the pivot, which goes into neither recursive call. Everything in this section is Equation (5.10) with a different k.

5.9.2 The best case: perfectly balanced splits

Suppose the pivot is always the median, so each partition splits the subarray into two halves of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$. Ignoring floors and ceilings, as Chapter 3 showed we may,

$$T(n) = 2T(n/2) + \Theta(n). \tag{5.11}$$

That is Merge Sort's recurrence exactly. Apply the Master Theorem to Equation (5.11): $a = 2$, $b = 2$, so $n^{\log_b a} = n^{\log_2 2} = n^1 = n$, and $f(n) = \Theta(n) = \Theta(n^{\log_b a})$. This is Case 2 with $k = 0$, giving

$$T(n) = \Theta(n \lg n).$$

Figure 5.18 is the recursion tree, which shows the same result without any theorem.

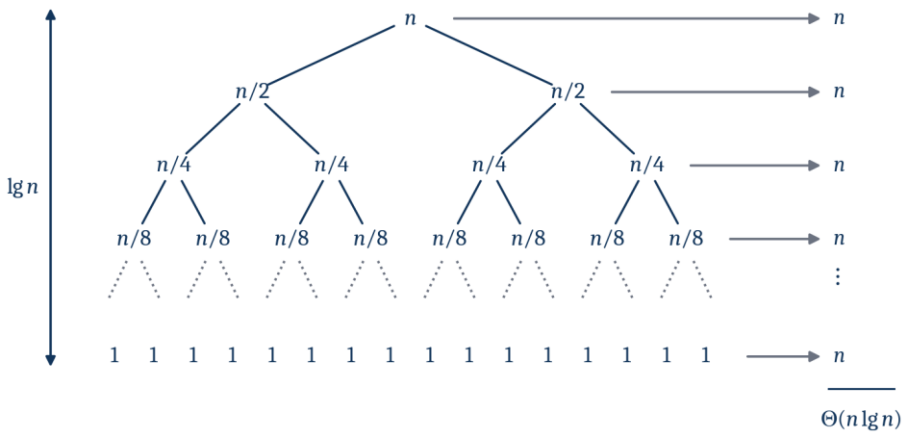


Figure 5.18 The best-case recursion tree for Quicksort. Each level's subproblem sizes halve — n, then $n/2$ twice, then $n/4$ four times — but the *total* work on each level stays n, because partitioning 2^d subarrays of size $n/2^d$ costs $2^d \times n/2^d = n$. There are $\lg n + 1$ levels, so the total is $\Theta(n \lg n)$.

Read **Figure 5.18** carefully, because the same picture explains every other case in this section. The arrows to the right of each level carry the level's total cost. Every one says n. The height of the tree is $\lg n$ because the sizes halve. Multiply: $n \times \lg n$.

5.9.3 The worst case: maximally unbalanced splits

Now suppose the pivot is always the smallest (or always the largest) element of the subarray. Then one side gets 0 elements and the other gets $n - 1$. Putting $k = 0$ into the general recurrence,

$$T(n) = T(0) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n), \tag{5.12}$$

since $T(0) = \Theta(1)$ is absorbed. **Figure 5.19** shows the derivation of the closed form in compact form.

$$\begin{aligned} T(n) &= T(1) + T(n-1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \sum_{k=1}^{n} \Theta(k) \\ &= \Theta\left(\sum_{k=1}^{n} k\right) \\ &= \Theta(n^2) \end{aligned}$$

Figure 5.19 The worst-case recurrence unrolled: $T(n) = T(1) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \Theta(n^2)$.

Let us unroll Equation (5.12) explicitly, one line at a time, because it is short and completely convincing:

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= [T(n-2) + c(n-1)] + cn \\ &= [T(n-3) + c(n-2)] + c(n-1) + cn \\ &\;\;\vdots \\ &= T(0) + c(1 + 2 + 3 + \cdots + (n-1) + n) \\ &= \Theta(1) + c \cdot \frac{n(n+1)}{2} \\ &= \Theta(n^2), \end{aligned}$$

using the arithmetic series $\sum_{k=1}^{n} k = n(n+1)/2$ from Chapter 3. **Figure 5.20** draws the same computation as a recursion tree.

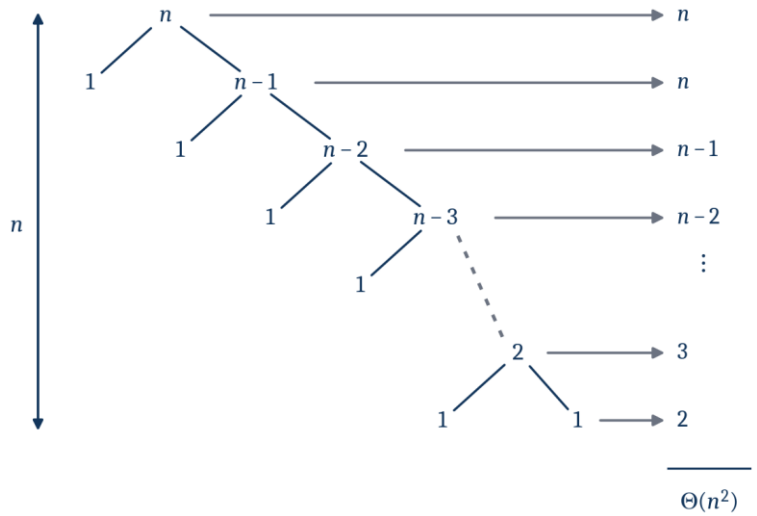


Figure 5.20 The worst-case recursion tree. Instead of branching, the tree degenerates into a single chain of length n — each node spawns one child of size one less and one child of size 0. The level costs are $n, n, n-1, n-2, \ldots, 2$, and summing them gives $\Theta(n^2)$.

The tree in **Figure 5.20** is the visual definition of the worst case: **a tree that is not a tree.** It has depth n instead of depth $\lg n$, and Quicksort has degenerated into something very like Selection Sort.

When does this actually happen? With Lomuto's partition and the last element as pivot, it happens when the array is already sorted, or sorted in reverse. On a sorted array, $A[r]$ is the largest element, so every element passes the test $A[j] \leq x$, every one joins the left region, and the partition returns $q = r$ — the extreme split, every time.

> **THE IRONY.** Already-sorted input is Insertion Sort's **best** case, $\Theta(n)$. It is naive Quicksort's **worst** case, $\Theta(n^2)$. And sorted or nearly-sorted input is not exotic: it is what you get from a database that returned rows in index order, a log file, a list of dates, or an array that someone has already sorted once. Never deploy a fixed-pivot Quicksort on data you did not generate yourself. Section 5.10 gives the fix.

An upper bound, proved by substitution. The recursion tree shows the worst case is at least $\Theta(n^2)$; to be rigorous we should also prove that it is never worse. Take the full worst-case recurrence, maximizing over all possible splits:

$$T(n) = \max_{0 \le k \le n-1} (T(k) + T(n-k-1)) + \Theta(n).$$

Guess $T(n) \le cn^2$ for some constant $c > 0$, and substitute:

$$\begin{aligned} T(n) &\le \max_{0 \le k \le n-1} (ck^2 + c(n-k-1)^2) + dn \\ &= c \cdot \max_{0 \le k \le n-1} (k^2 + (n-k-1)^2) + dn. \end{aligned}$$

The expression $k^2 + (n-k-1)^2$ is an upward-opening parabola in k, so on the interval $[0, n-1]$ its maximum is at one of the endpoints. At $k = 0$ it is $(n-1)^2$; at $k = n-1$ it is also $(n-1)^2$. So

$$\begin{aligned} T(n) \le c(n-1)^2 + dn = cn^2 - 2cn + c + dn = cn^2 - c(2n-1) + dn \\ \le cn^2, \end{aligned}$$

provided $c(2n-1) \ge dn$, which holds for all $n \ge 1$ as soon as we choose $c \ge d$. So $T(n) = O(n^2)$, and combined with the tree's $\Omega(n^2)$ we have $T(n) = \Theta(n^2)$ in the worst case. ■

5.9.4 The balanced-split intuition, and the 9: 1 split

Here is the fact that saves Quicksort, and it is genuinely surprising the first time you meet it.

Suppose the partition is *always* lopsided, but lopsided by a fixed ratio — say it always puts 10% of the elements on one side and 90% on the other. That sounds bad. The recurrence is

$$T(n) = T(n/10) + T(9n/10) + \Theta(n). \tag{5.13}$$

Figure 5.21 is the recursion tree of Equation (5.13).

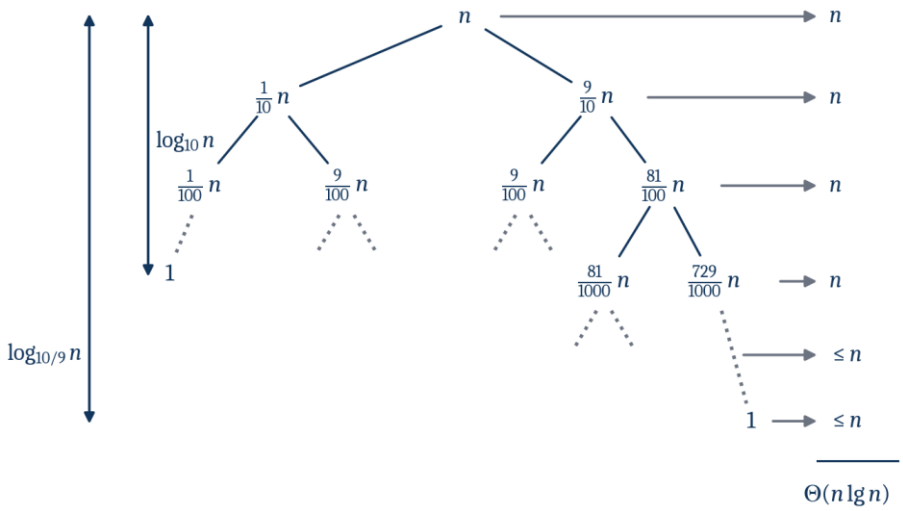


Figure 5.21 The recursion tree for a constant 1: 9 split. The left spine, whose sizes shrink by a factor of ten each level, bottoms out after $\log_{10} n$ levels. The right spine, whose sizes shrink only by a factor of 10/9, survives to depth $\log_{10/9} n$. Between those two depths some levels are 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 $\Theta(n\lg n)$.

Work through **Figure 5.21** node by node. The root is size n and costs n. Its children are $n/10$ and $9n/10$; together they cost $n/10 + 9n/10 = n$. Their children are $n/100$, $9n/100$, $9n/100$ and $81n/100$; together $\frac{1+9+9+81}{100}n = n$. **Every complete level costs exactly** n, because partitioning never loses an element.

So the only question is: how many levels are there? Two answers, because the tree is lopsided.

- Following the *shortest* path — always to the small side — the sizes are $n, n/10, n/100, \ldots$, reaching 1 after $\log_{10} n$ steps.
- Following the *longest* path — always to the big side — the sizes are $n, \frac{9}{10}n, \left(\frac{9}{10}\right)^2 n, \ldots$, reaching 1 when $\left(\frac{9}{10}\right)^d n = 1$, that is when $d = \log_{10/9} n$.

Every level above depth $\log_{10} n$ is complete and costs exactly n; every level below costs at most n. So

$$n\log_{10} n \;\le\; T(n) \;\le\; n\log_{10/9} n + \Theta(n).$$

Now convert both logarithms to base 2 using $\log_b n = \lg n / \lg b$:

$$\log_{10} n = \frac{\lg n}{\lg 10} \approx \frac{\lg n}{3.32} \approx 0.30 \lg n, \qquad \log_{10/9} n = \frac{\lg n}{\lg(10/9)} \approx \frac{\lg n}{0.152} \approx 6.58 \lg n.$$

So $T(n)$ lies between about $0.30\, n\lg n$ and about $6.58\, n\lg n$. Both ends are $\Theta(n\lg n)$. The lopsidedness has cost us a **constant factor of about twenty-two** — and nothing else.

> **THE CRUCIAL FACT. Any** split in a constant ratio gives $\Theta(n\lg n)$. Fifty-fifty, ninety-ten, ninety-nine to one — it makes no difference to the asymptotic answer, only to the constant. The recursion tree has depth $\log_{1/\alpha} n = \Theta(\lg n)$ for any fixed fraction α, and each level costs n. **Only a split that is not in constant ratio — the degenerate** $1:(n-1)$ **split, where the small side has a fixed number of elements rather than a fixed fraction — produces** $\Theta(n^2)$**.** That is why Quicksort is so hard to make slow by accident: you have to be unlucky at essentially every single partition, not merely at most of them.

Let us make that concrete with numbers. For $n = 1{,}000{,}000$:

- A 99: 1 split every time: depth $\log_{100/99} 10^6 = \lg 10^6 / \lg(100/99) \approx 19.93/0.0145 \approx 1{,}375$ levels. Total work about 1.4×10^9 operations — a second or two.
- The degenerate $1:(n-1)$ split every time: depth 10^6, total work 5×10^{11} operations — about eight minutes.

A factor of more than three hundred separates "consistently terrible split" from "degenerate split". That gap is Quicksort's safety margin.

5.9.5 The average case, argued two ways

The best case is $\Theta(n\lg n)$ and the worst is $\Theta(n^2)$. Which do we get in practice? The average case — and it is very close to the best case.

First argument: bad splits are absorbed by the good ones that follow. This is the argument to hold in your head. Suppose the luck alternates perfectly: one level splits right down the middle (lucky), the next level splits 1: $(n-1)$ (unlucky), the next is lucky again, and so on. Write $L(n)$ for the cost when the top level is lucky and $U(n)$ for the cost when it is unlucky. **Figure 5.22** sets out the derivation.

$$L(n) = 2U(n/2) + \Theta(n) \qquad \text{lucky}$$

$$U(n) = L(n-1) + \Theta(n) \qquad \text{unlucky}$$

we consequently get

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

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 $L(n) = 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) = 2L(n/2-1) + \Theta(n) = \Theta(n\lg n)$.

The two recurrences are

$$L(n) = 2\,U(n/2) + \Theta(n) \qquad \text{(lucky: two halves, each then unlucky)}$$

$$U(n)$$
$$= L(n-1) + \Theta(n) \qquad \text{(unlucky: one side empty, the other then lucky)}$$

Substitute the second into the first:

$$\begin{aligned} L(n) &= 2\,U(n/2) + \Theta(n) \\ &= 2[\,L(n/2-1) + \Theta(n/2)\,] + \Theta(n) \\ &= 2\,L(n/2-1) + \Theta(n) + \Theta(n) \\ &= 2\,L(n/2-1) + \Theta(n). \end{aligned}$$

The last line is again the Merge Sort recurrence — the -1 inside is a lower-order perturbation that Chapter 3 told us we may ignore — so $L(n) = \Theta(n\lg n)$.

Figure 5.23 and **Figure 5.24** put the same point in a picture, and they are worth studying side by side.

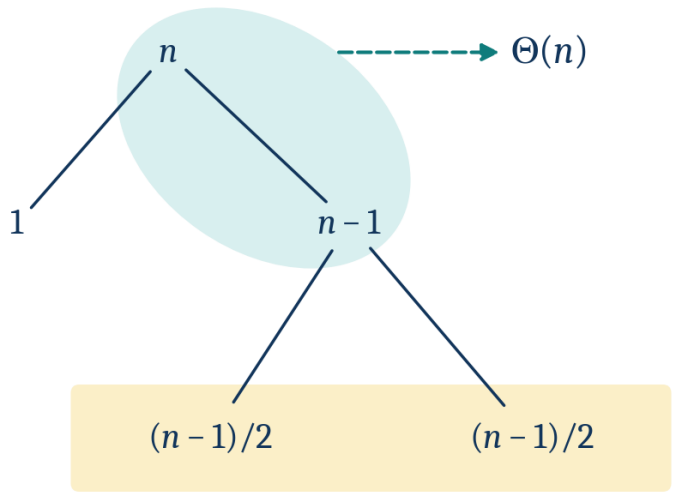


Figure 5.23 An unlucky split followed by a lucky one. The unlucky partition of n elements yields a piece of size 1 and a piece of size $n-1$; the lucky partition of that piece yields two pieces of size $(n-1)/2$. The shaded region shows the combined cost, $\Theta(n) + \Theta(n-1) = \Theta(n)$.

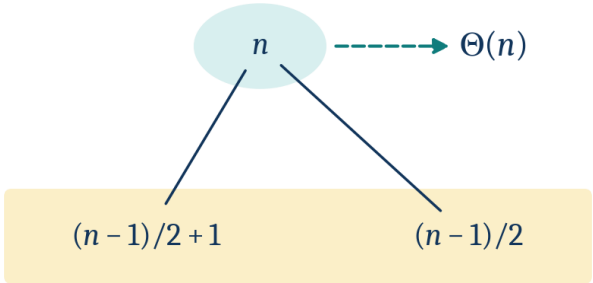


Figure 5.24 A single lucky split, for comparison: one partition of n elements, costing $\Theta(n)$, yielding two pieces of size about $(n-1)/2$.

Compare **Figure 5.23** with **Figure 5.24**. They end in the same place — two subproblems of size about $(n-1)/2$ — and they cost the same to within a constant factor: two levels of $\Theta(n)$ work instead of one. So **an unlucky split does not double the running time; it merely adds one extra level.** The cost of bad luck is a constant factor, not a change of growth rate. That is what "absorbed" means.

Second argument: the expected number of comparisons, counted exactly. The alternating argument is intuition. Here is the real theorem, and it is one of the prettiest arguments in the whole of algorithm analysis.

Rename the array elements $z_1, z_2, \ldots, z_n$ in *sorted* order, so z_1 is the smallest and z_n the largest. Write $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ for the set of elements whose sorted positions lie between i and j inclusive.

Now make one observation that does all the work:

> **THE KEY OBSERVATION.** Two elements z_i and z_j are compared **at most once** in the entire run of Quicksort. They are compared only when one of them is the pivot, and a pivot is compared with everything in its subarray and then removed from the recursion for ever.

So the total number of comparisons is just the number of pairs that ever get compared. Define the indicator random variable

$$X_{ij} = \begin{cases} 1 & \text{if } z_i \text{ is compared with } z_j \text{ at some point,} \\ 0 & \text{otherwise,} \end{cases}$$

so the total comparison count is $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$, and by linearity of expectation

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared with } z_j\}.$$

When are z_i and z_j compared? Follow the recursion. As long as no pivot has been chosen from Z_{ij}, all of Z_{ij} stays together in the same subarray — because any pivot outside Z_{ij} is either smaller than every element of Z_{ij} or larger than every element of it, and so sends the whole set to one side. The moment some element of Z_{ij} is chosen as a pivot, one of three things happens:

- the pivot is z_i — then z_i is compared with everything in the subarray, including z_j. **Compared.**
- the pivot is z_j — same, by symmetry. **Compared.**
- the pivot is some z_k with $i < k < j$ — then z_i goes to the left piece, z_j goes to the right piece, and they are never in the same subarray again. **Never compared.**

So z_i and z_j are compared **exactly when the first pivot drawn from Z_{ij} is z_i or z_j.** The set Z_{ij} has $j - i + 1$ elements, and — with a randomly chosen pivot, which is the subject of Section 5.10 — each is equally likely to be the first one chosen. Hence

$$\Pr\{z_i \text{ is compared with } z_j\} = \frac{2}{j - i + 1}. \tag{5.14}$$

Substituting Equation (5.14) into the double sum, and writing $k = j - i$:

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = 2 \sum_{i=1}^{n-1} H_n,$$

where $H_n = \sum_{k=1}^{n} 1/k$ is the harmonic number from Chapter 3. So

$$\mathbb{E}[X] < 2(n - 1)H_n = O(n\lg n),$$

because $H_n = \ln n + O(1)$ and $\ln n = \lg n \cdot \ln 2 \approx 0.693 \lg n$. A more careful evaluation of the double sum, keeping the lower-order terms, gives the exact answer [8]

$$\mathbb{E}[X] = 2(n + 1)H_n - 4n \approx 2n\ln n - 2.85n \approx 1.386\, n\lg n. \tag{5.15}$$

> **WHAT $1.386\, n\lg n$ MEANS.** The best possible comparison sort needs about $n\lg n$ comparisons (Section 5.11). Randomized Quicksort averages about 1.39 times that. Merge Sort uses about $n\lg n$. So Quicksort makes roughly **thirty-nine percent more comparisons** than Merge Sort — and is still faster in practice, because a comparison is cheap and a cache miss is not. Section 5.12.3 measures this.

5.9.6 Checking the theory against a machine

Listing 5.9 counts the actual comparisons Quicksort performs on three kinds of input, and compares them against the two formulas we have derived: $n(n-1)/2$ for the worst case and $2(n+1)H_n - 4n$, from Equation (5.15), for the average.

Listing 5.9 Counting Quicksort's comparisons on sorted and shuffled input, with and without a random pivot, against the two theoretical formulas.

```
// How many comparisons does quicksort really make? Fixed pivot vs. random pivot.
#include <iostream>
#include <iomanip>
#include <vector>
#include <numeric>
#include <algorithm>
#include <random>
#include <cmath>
using namespace std;

long long comparisons;
mt19937 rng(20260919);

int partition(vector<int>& A, int p, int r) {
    int x = A[r], i = p - 1;
    for (int j = p; j <= r - 1; ++j) {
        ++comparisons;                          // this is THE comparison of the algorithm
        if (A[j] <= x) swap(A[++i], A[j]);
    }
    swap(A[i + 1], A[r]);
    return i + 1;
}

int randomizedPartition(vector<int>& A, int p, int r) {
    uniform_int_distribution<int> pick(p, r);
    swap(A[r], A[pick(rng)]);                   // a uniformly random element becomes the
pivot
    return partition(A, p, r);
}

void quicksort(vector<int>& A, int p, int r, bool randomized) {
    if (p >= r) return;
    int q = randomized ? randomizedPartition(A, p, r) : partition(A, p, r);
    quicksort(A, p, q - 1, randomized);
    quicksort(A, q + 1, r, randomized);
}

int main() {
    cout << fixed << setprecision(2);
    cout << "        n |  sorted input, fixed pivot |  random input, fixed pivot |"
            "  sorted input, random pivot |   n(n-1)/2 |  2(n+1)H_n-4n\n";
    for (int n : {100, 500, 1000, 2000, 4000}) {
        vector<int> sorted(n);
        iota(sorted.begin(), sorted.end(), 1);   // 1, 2, 3, ..., n -- already in order
```

Listing 5.9 (Continued) *Counting Quicksort's comparisons on sorted and shuffled input, with and without a random pivot, against the two theoretical formulas.*

```
        vector<int> shuffled = sorted;
        shuffle(shuffled.begin(), shuffled.end(), rng);

        vector<int> a = sorted;   comparisons = 0; quicksort(a, 0, n - 1, false);
        long long worst = comparisons;

        vector<int> b = shuffled; comparisons = 0; quicksort(b, 0, n - 1, false);
        long long avg = comparisons;

        vector<int> c = sorted;   comparisons = 0; quicksort(c, 0, n - 1, true);
        long long rnd = comparisons;

        double half = (double)n * (n - 1) / 2.0;
        double H = 0;                            // the n-th harmonic number, summed
directly
        for (int k = 1; k <= n; ++k) H += 1.0 / k;
        double pred = 2.0 * (n + 1) * H - 4.0 * n;   // the exact expected comparison count
        cout << setw(9) << n << " | " << setw(26) << worst << " | " << setw(26) << avg
             << " | " << setw(27) << rnd << " | " << setw(10) << half
             << " | " << setw(12) << pred << "\n";
    }
    return 0;
}
```

The output:

```
        n |  sorted input, fixed pivot |  random input, fixed pivot |  sorted input, random
pivot |   n(n-1)/2 |  2(n+1)H_n-4n
      100 |                       4950 |                        623 |
634 |    4950.00 |       647.85
      500 |                     124750 |                       5146 |
5144 |  124750.00 |      4806.41
     1000 |                     499500 |                      10452 |
11176 |  499500.00 |     10985.91
     2000 |                    1999000 |                      22672 |
23687 | 1999000.00 |     24729.83
     4000 |                    7998000 |                      55683 |
57639 | 7998000.00 |     54988.87
```

Three things to read out of that table.

One. The "sorted input, fixed pivot" column is *exactly* $n(n-1)/2$ — 4950 against 4950.00, 499500 against 499500.00, every row. Not approximately; exactly. That is the worst case, achieved, and it confirms the derivation of Section 5.9.3 down to the last comparison. Every pair of elements is compared, which is $\binom{n}{2} = n(n-1)/2$ pairs.

Two. The two average-case columns track the formula $2(n+1)H_n - 4n$ closely — 55,683 and 57,639 against a prediction of 54,989 at $n = 4000$, within about five percent. Randomness means a single run will not land on the expectation exactly, but it lands nearby.

Three, and most important. Compare the first column with the third at $n = 4000$: 7,998,000 comparisons against 57,639. **The same input, the same algorithm, one line different** — and it runs one hundred and thirty-eight times faster. That one line is the subject of the next section.

5.10 Randomized Quicksort

5.10.1 The fix

The problem with `PARTITION` as written is that the pivot is chosen by *position* — always the last element. An adversary who knows this can hand you an input that makes every partition degenerate. And you do not need a malicious adversary: "already sorted" is a perfectly ordinary input that has the same effect.

The fix is one swap, and **Pseudocode 5.12** shows it.

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)
```

Line 1 picks a random index; line 2 moves that element into the pivot slot; line 3 proceeds exactly as before. Two extra lines, and everything changes.

5.10.2 What randomization actually buys

It is important to be precise here, because students routinely overclaim.

> **WHAT RANDOMIZATION DOES AND DOES NOT DO.** 1. **It makes the running time independent of the input ordering.** The pivot is no longer determined by where elements sit, so no arrangement of the data is inherently bad. The randomness comes from the algorithm's coin flips, not from assumptions about the input. 2. **No specific input triggers the worst case.** There is no "sorted array" problem any more, and an adversary who can see your data cannot construct a bad case, because they cannot see your random numbers. 3. **The worst case can still occur — by bad luck.** Randomization does not remove the $\Theta(n^2)$ branch; it makes it astronomically unlikely. 4. **The expected running time is $O(n\lg n)$, with about $1.39\,n\lg n$ comparisons** — the result derived in Section 5.9.5, which assumed exactly this uniform random pivot.

Point 3 deserves a number. For every partition to be degenerate, the randomly chosen pivot must be the minimum or maximum of its subarray, every single time. At the top level that has probability $2/n$; at the next $2/(n-1)$; and so on. The probability of the full disaster is about

$$\frac{2}{n}\cdot\frac{2}{n-1}\cdots\frac{2}{2}=\frac{2^{n-1}}{n!}. \tag{5.16}$$

For $n=100$ that is roughly 10^{-128}. For comparison, the number of atoms in the observable universe is about 10^{80}. Picking one particular atom out of the whole observable universe by blind chance is more likely than this disaster by a factor of about 10^{48}.

> **THE DEEPER POINT.** Before randomization, Quicksort's running time was a function of the input: some inputs were fast, some were slow, and which was which was a fixed, discoverable

> property. After randomization, the running time is a function of the algorithm's own coin flips, and **every** input has the same distribution of running times. We have not made the algorithm faster; we have made it *predictable*. In a production system that difference is worth more than speed.

5.10.3 Other pivot strategies

Randomization is not the only defense, and real libraries combine several.

- **Median-of-three.** Take the median of $A[p]$, $A[\lfloor(p+r)/2\rfloor]$ and $A[r]$ and use that as the pivot. It costs three comparisons and defeats sorted and reverse-sorted input outright, since the middle element of a sorted array is its median. It is not immune to a deliberately crafted adversarial input, but it is very cheap and very effective on real data.
- **Dual-pivot (Yaroslavskiy, 2009).** Choose two pivots $x \leq y$ and partition into three regions: $<x$, between x and y, and $>y$. Fewer passes over the data for the same amount of sorting. This is what Java's `Arrays.sort` uses for primitive types; the JDK 25 documentation describes it as offering $O(n\lg n)$ performance on all data sets [9].
- **Introsort (Musser, 1997)** [10]**.** Start with Quicksort. Track the recursion depth; if it exceeds about $2\lg n$ — the sign that the splits are going badly — switch that subproblem to Heapsort, which cannot be made slow. Also stop recursing on subarrays smaller than about sixteen elements and finish with a single Insertion Sort pass over the whole array. This is what C++'s `std::sort` does, and it is the most practically important algorithm in this chapter: it has Quicksort's speed, Heapsort's guarantee, and Insertion Sort's low overhead on small inputs, all at once.
- **Three-way partitioning** — the *Dutch national flag* problem [11]. If the data has many duplicate keys, partition into $<x$, $=x$, $>x$ and recurse only on the outer two. An array of a million elements with only ten distinct values sorts in $\Theta(n)$ time this way.

> **IN A CONTEST, AND IN PRODUCTION.** Use the language's built-in sort. C++'s `std::sort` is Introsort; Java uses dual-pivot Quicksort for primitives and TimSort for objects; Python uses Timsort, whose rule for merging runs was replaced in Python 3.11 by the near-optimal Powersort policy of Munro and Wild [12]. They are faster than anything you will write in the time available, and they are correct [4], [5]. Learn the analysis in this chapter because it tells you *what the library is doing* and *when it will be slow* — not so that you can re-implement it under time pressure.

5.10.4 Tail recursion and the stack

There is one more practical hazard, and it is not about time at all: it is about memory.

Every recursive call consumes a **stack frame** — space for the parameters, the return address, and the local variables. Quicksort's recursion depth is the height of its recursion tree. In the best case that is $\Theta(\lg n)$, a handful of frames. In the worst case it is $\Theta(n)$ — and a million nested calls will exhaust the call stack and crash the program, long before the $\Theta(n^2)$ time becomes an issue.

The fix uses a property of line 4 of `QUICKSORT` in **Pseudocode 5.11**: it is a **tail call**, the last thing the procedure does. A tail call needs no new stack frame — instead of recursing, we can simply update the parameters and loop. **Pseudocode 5.13** does exactly that.

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):
 while p < r
     q ← PARTITION(A, p, r)
     if (q − p) < (r − q)                    // the LEFT piece is smaller
         QUICKSORT-TAIL(A, p, q − 1)         //   recurse on it ...
         p ← q + 1                           //   ... and loop on the right
     else                                    // the RIGHT piece is smaller
         QUICKSORT-TAIL(A, q + 1, r)         //   recurse on it ...
         r ← q − 1                           //   ... and loop on the left
```

Two changes, and both are needed:

1. **Only one of the two calls is a real recursion**; the other becomes a loop iteration. That alone halves the number of frames.
2. **The recursion is always on the *smaller* piece.** This is the change that matters. Each real recursive call at least halves the size of the subproblem — because the smaller of two pieces is at most half of their total — so the depth cannot exceed $\lg n$, no matter how badly the partitions go.

> **THEOREM.** `QUICKSORT-TAIL` uses $O(\lg n)$ stack space in the worst case. **Proof.** Every recursive call is made on a subarray of size at most half that of its caller, since it is the smaller of two pieces whose sizes sum to at most the caller's size minus one. Starting from n, halving at every level of genuine recursion, the depth d satisfies $n/2^d \geq 1$, hence $d \leq \lg n$. Each frame is $O(1)$, so the total is $O(\lg n)$. ■

Listing 5.10 measures the difference.

Listing 5.10 Measuring the recursion depth of naive and tail-recursive Quicksort on sorted and shuffled input.

```
// Recursion depth: naive quicksort against the tail-recursive version that always
// recurses on the SMALLER side first.
#include <iostream>
#include <iomanip>
#include <vector>
#include <numeric>
#include <algorithm>
#include <random>
using namespace std;

int depthNow, depthMax;

struct DepthGuard {                              // counts how deep the C++ call stack goes
    DepthGuard()  { if (++depthNow > depthMax) depthMax = depthNow; }
    ~DepthGuard() { --depthNow; }
};

int partition(vector<int>& A, 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 quicksortNaive(vector<int>& A, int p, int r) {
    DepthGuard g;
    if (p >= r) return;
    int q = partition(A, p, r);
    quicksortNaive(A, p, q - 1);
    quicksortNaive(A, q + 1, r);
}

void quicksortTail(vector<int>& A, int p, int r) {
    DepthGuard g;
    while (p < r) {
        int q = partition(A, p, r);
        if (q - p < r - q) {                     // the LEFT side is smaller: recurse on it
            quicksortTail(A, p, q - 1);
            p = q + 1;                           // and loop on the right side instead
        } else {                                 // the RIGHT side is smaller: recurse on
it
            quicksortTail(A, q + 1, r);
            r = q - 1;                           // and loop on the left side instead
        }
    }
}
```

Listing 5.10 (Continued) *Measuring the recursion depth of naive and tail-recursive Quicksort on sorted and shuffled input.*

```
int main() {
    mt19937 rng(20260919);
    cout << "        n | sorted: naive | sorted: tail | shuffled: naive | shuffled: tail |
lg n\n";
    for (int n : {1000, 4000, 16000, 64000}) {
        vector<int> sorted(n); iota(sorted.begin(), sorted.end(), 1);
        vector<int> shuffled = sorted; shuffle(shuffled.begin(), shuffled.end(), rng);

        vector<int> a = sorted;   depthNow = depthMax = 0; quicksortNaive(a, 0, n - 1); int
sn = depthMax;
        vector<int> b = sorted;   depthNow = depthMax = 0; quicksortTail (b, 0, n - 1); int
st = depthMax;
        vector<int> c = shuffled; depthNow = depthMax = 0; quicksortNaive(c, 0, n - 1); int
rn = depthMax;
        vector<int> d = shuffled; depthNow = depthMax = 0; quicksortTail (d, 0, n - 1); int
rt = depthMax;

        int lg = 0; { int m = n; while (m > 1) { m >>= 1; ++lg; } }
        cout << setw(9) << n << " | " << setw(13) << sn << " | " << setw(12) << st
             << " | " << setw(15) << rn << " | " << setw(14) << rt
             << " | " << setw(6) << lg << "\n";
    }
    return 0;
}
```

The output:

```
        n | sorted: naive | sorted: tail | shuffled: naive | shuffled: tail |  lg n
     1000 |          1000 |            2 |               24 |               7 |      9
     4000 |          4000 |            2 |               31 |               8 |     11
    16000 |         16000 |            2 |               32 |              10 |     13
    64000 |         64000 |            2 |               39 |              11 |     15
```

The naive version on sorted input reaches a depth of exactly n: sixty-four thousand nested stack frames. On a typical eight-megabyte stack with frames of a hundred bytes or so, that is already close to the limit, and a million-element sorted array would certainly crash. The tail-recursive version reaches depth **two** on the same input — one genuine recursion into an empty left piece, then a loop — and never exceeds $\lg n$ on shuffled input either: $7 \le 9$, $8 \le 11$, $10 \le 13$, $11 \le 15$, exactly as the theorem promised.

> **RECURSE ON THE SMALLER SIDE.** If you take one implementation habit from this chapter, take this one. It costs a single comparison per partition and it converts a crash into a non-event. Every production Quicksort does it.

5.11 The $\Omega(n\lg n)$ Lower Bound for Comparison Sorts

We promised in Section 5.1.3 to justify the claim that $n\lg n$ is a floor, not merely the best we have managed so far. Here is the argument. It is short, it is complete, and it is one of the few places in this course where we prove that something is **impossible**.

5.11.1 What counts as a comparison sort

Every algorithm in this chapter, and in Chapters 2 and 3, learns about the input in exactly one way: it asks questions of the form "is $a_i \le a_j$?" It never looks at the *values* — never uses the fact that a key is the number 47 rather than the string `"pineapple"`. Such an algorithm is called a **comparison sort**.

Insertion Sort, Selection Sort, Bubble Sort, Merge Sort, Heapsort and Quicksort are all comparison sorts. Their control flow is determined entirely by the answers to yes/no comparisons.

5.11.2 The decision tree

Fix the algorithm and fix the input size n. Now trace what the algorithm does, as a tree:

- Each **internal node** is labeled with a comparison $a_i : a_j$ that the algorithm performs at that point.
- Its two children correspond to the two possible answers, $\leq$ and $>$.
- Each **leaf** is labeled with the permutation the algorithm outputs when it reaches that leaf.

This is the algorithm's **decision tree** for inputs of size n. Running the algorithm on a particular input means walking from the root to a leaf, turning left or right at each node according to the answer.

Two facts follow immediately.

Fact 1: the tree must have at least $n!$ leaves. There are $n!$ possible orderings of n distinct elements, and a correct sorting algorithm must produce a different output permutation for each of them. If two different orderings led to the same leaf, the algorithm would output the same permutation for both — and at most one of those outputs can be correct.

Fact 2: the worst-case number of comparisons is the height of the tree. The number of comparisons on a given input is the number of internal nodes on the path from the root to its leaf, and the worst case over all inputs is the longest such path.

5.11.3 The proof

THEOREM. Any comparison sort requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof. Let the decision tree have height h and ℓ leaves.

A binary tree of height h has at most 2^h leaves — one leaf for each of the at most 2^h distinct root-to-leaf paths of length h. By Fact 1, $\ell \geq n!$. Therefore

$$n! \;\leq\; \ell \;\leq\; 2^h. \tag{5.17}$$

Take lg of both ends of Equation (5.17), which preserves the inequality because lg is increasing:

$$h \;\geq\; \lg(n!).$$

Now bound $\lg(n!)$ from below. Stirling's approximation from Chapter 3 says $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + \Theta(1/n))$, so

$$\lg(n!) \;\geq\; \lg\left(\left(\frac{n}{e}\right)^n\right) \;=\; n\lg n - n\lg e \;=\; n\lg n - 1.443n \;=\; \Omega(n\lg n). \tag{5.18}$$

Hence $h = \Omega(n\lg n)$: some input forces the algorithm to make at least that many comparisons. ■

If you would rather avoid Stirling, here is an elementary version that gives a slightly weaker constant. Of the n factors in $n! = 1 \cdot 2 \cdots n$, the largest $n/2$ of them are each at least $n/2$. So $n! \geq (n/2)^{n/2}$, and

$$\lg(n!) \geq \frac{n}{2}\lg\frac{n}{2} = \frac{n}{2}\lg n - \frac{n}{2} = \Omega(n\lg n).$$

5.11.4 What the bound does and does not forbid

READ THE HYPOTHESIS. The theorem says *comparison* sorts cannot beat $\Omega(n\lg n)$. It says nothing about algorithms that look at the keys themselves. Counting Sort, Radix Sort and Bucket Sort, treated in full in Chapter 8 of CLRS [3], sort in $\Theta(n)$ time by using the values of the keys as array indices rather than merely comparing them. They do not contradict the theorem; they fall outside its hypothesis. The cost is that they need to know something about the keys in advance: that they are small integers, or that they are drawn from a known distribution.

Listing 5.11 makes the argument concrete for small n, and tabulates the bound.

Listing 5.11 The decision-tree bound computed for small n: the number of leaves required, and the resulting minimum height.

```
// The decision-tree bound, made concrete: how many leaves must a comparison sort have,
// and how tall must a binary tree with that many leaves be?
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
#include <cmath>
using namespace std;

long long factorial(int n) { long long f = 1; for (int i = 2; i <= n; ++i) f *= i; return
f; }

int main() {
    cout << "All 3! = 6 orderings of three distinct keys a, b, c, and the comparisons\n"
            "a comparison sort must make to tell each one apart:\n\n";
    vector<int> p = {1, 2, 3};
    do {
        cout << "  permutation (" << p[0] << "," << p[1] << "," << p[2] << ")"
             << "  a<b? " << (p[0] < p[1] ? "yes" : "no ")
             << "  b<c? " << (p[1] < p[2] ? "yes" : "no ")
             << "  a<c? " << (p[0] < p[2] ? "yes" : "no ") << "\n";
    } while (next_permutation(p.begin(), p.end()));

    cout << "\nSix outcomes need six leaves. A binary tree of height h has at most 2^h\n"
            "leaves, so h >= lg 6 = " << fixed << setprecision(4) << log2(6.0)
         << ", and since h is an integer, h >= 3.\n\n";

    cout << "    n |                  n! | lg(n!) | ceil(lg n!) | n lg n\n";
    for (int n : {2, 3, 4, 5, 8, 10, 12, 16, 20}) {
        long long f = factorial(n);
        double lgf = 0;                          // lg(n!) = sum of lg k, summed to avoid
overflow
        for (int k = 2; k <= n; ++k) lgf += log2((double)k);
        cout << setw(5) << n << setw(22) << f
             << setw(9) << setprecision(1) << lgf
             << setw(14) << (long long)ceil(lgf)
             << setw(9) << n * log2((double)n) << setprecision(4) << "\n";
    }
    return 0;
}
```

The output:

```
All 3! = 6 orderings of three distinct keys a, b, c, and the comparisons
a comparison sort must make to tell each one apart:

  permutation (1,2,3)  a<b? yes  b<c? yes  a<c? yes
  permutation (1,3,2)  a<b? yes  b<c? no   a<c? yes
  permutation (2,1,3)  a<b? no   b<c? yes  a<c? yes
  permutation (2,3,1)  a<b? yes  b<c? no   a<c? no
  permutation (3,1,2)  a<b? no   b<c? yes  a<c? no
  permutation (3,2,1)  a<b? no   b<c? no   a<c? no

Six outcomes need six leaves. A binary tree of height h has at most 2^h
leaves, so h >= lg 6 = 2.5850, and since h is an integer, h >= 3.

    n |                  n! | lg(n!) | ceil(lg n!) | n lg n
    2                     2      1.0             1      2.0
    3                     6      2.6             3      4.8
    4                    24      4.6             5      8.0
    5                   120      6.9             7     11.6
    8                 40320     15.3            16     24.0
   10               3628800     21.8            22     33.2
   12             479001600     28.8            29     43.0
   16         20922789888000     44.3            45     64.0
   20    2432902008176640000     61.1            62     86.4
```

Read the first block: no two of the six permutations agree on all three comparisons, so three yes/no questions are exactly enough to tell them apart — and two would not be, since two questions have only four possible answers and there are six cases. The bound is achieved.

Read the table: for $n = 20$, no comparison sort can always finish in fewer than 62 comparisons. The last column shows $n\lg n = 86.4$, so the bound $\lceil \lg(n!) \rceil$ is roughly $0.7\, n\lg n$ — the constant is less than one, but the growth rate is the same, and that is what $\Omega(n\lg n)$ claims.

> **WHAT THIS MEANS FOR THE CHAPTER.** Merge Sort, Heapsort and randomized Quicksort all run in $\Theta(n\lg n)$. The theorem says nothing can do better by comparisons. So these three algorithms are **asymptotically optimal** — and the only remaining questions are about constants, memory, guarantees and cache behavior. Those questions are the subject of the next section, and they are not small questions: they are what decides which algorithm actually ships.

5.12 Choosing a Sort: the Honest Comparison

5.12.1 The complete landscape

Table 5.7 is the whole course so far in one table.

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

Two entries in that table are easy to misread.

Quicksort's space is $\lg n$**, not** 1**.** Quicksort does not allocate any array, so it is in-place in the usual sense — but the recursion itself consumes stack, $\Theta(\lg n)$ frames' worth when written as in Section 5.10.4. That is the honest figure.

Quicksort's worst case really is n^2**.** The table does not lie about this, and the n^2 does not go away with randomization. What randomization changes is the *probability*, not the possibility.

5.12.2 Stability, and why anyone cares

We defined stability in Section 5.6.5: a sort is stable if elements with equal keys stay in their original relative order. Why does it matter?

Because real records have more than one field, and real users sort them more than once. Suppose a laundry's order list is sorted by customer name, and the operator then sorts it by pickup date. With a **stable** sort, orders due on the same date remain in name order — a two-key sort for the price of two one-key sorts. With an **unstable** sort, the name ordering is destroyed and the result is arbitrary within each date.

Stability is also what makes Radix Sort possible: it sorts by the least significant digit first, then the next, relying on each pass preserving the work of the previous ones.

Listing 5.12 demonstrates the property on a tiny input where you can check every element by eye.

Listing 5.12 A stability test. Each record carries a tag showing where it started, so reordering among equal keys is visible.

```
// Stability: does the sort keep equal keys in their original relative order?
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

struct Rec { int key; string tag; };             // tag records where the item started

void show(const string& label, const vector<Rec>& v) {
    cout << label;
    for (const Rec& r : v) cout << r.key << r.tag << " ";
    cout << "\n";
}

void insertionSort(vector<Rec>& A) {
    for (size_t j = 1; j < A.size(); ++j) {
        Rec key = A[j]; size_t i = j;
        while (i > 0 && A[i - 1].key > key.key) { A[i] = A[i - 1]; --i; }   // > , not >=
        A[i] = key;
    }
}
void mergeSortRec(vector<Rec>& A, vector<Rec>& b, int p, int r) {
    if (p >= r) return;
    int q = p + (r - p) / 2;
    mergeSortRec(A, b, p, q); mergeSortRec(A, b, q + 1, r);
    int i = p, j = q + 1, k = p;
    while (i <= q && j <= r) b[k++] = (A[i].key <= A[j].key) ? A[i++] : A[j++];   // <=
keeps left first
    while (i <= q) b[k++] = A[i++];
    while (j <= r) b[k++] = A[j++];
    for (int t = p; t <= r; ++t) A[t] = b[t];
}
void mergeSort(vector<Rec>& A) { vector<Rec> b(A.size()); mergeSortRec(A, b, 0,
(int)A.size() - 1); }

void siftDown(vector<Rec>& A, int i, int m) {
    while (true) {
        int l = 2 * i + 1, r = 2 * i + 2, big = i;
        if (l < m && A[l].key > A[big].key) big = l;
        if (r < m && A[r].key > A[big].key) big = r;
        if (big == i) return;
        swap(A[i], A[big]); i = big;
    }
}
void heapSort(vector<Rec>& A) {
    int m = (int)A.size();
    for (int i = m / 2 - 1; i >= 0; --i) siftDown(A, i, m);
```

Listing 5.12 (Continued) *A stability test. Each record carries a tag showing where it started, so reordering among equal keys is visible.*

```
    for (int i = m - 1; i >= 1; --i) { swap(A[0], A[i]); siftDown(A, 0, i); }
}

void quickSort(vector<Rec>& A, int p, int r) {
    if (p >= r) return;
    int x = A[r].key, i = p - 1;
    for (int j = p; j <= r - 1; ++j) if (A[j].key <= x) swap(A[++i], A[j]);
    swap(A[i + 1], A[r]);
    quickSort(A, p, i); quickSort(A, i + 2, r);
}

int main() {
    vector<Rec> original = {{3,"a"},{1,"b"},{3,"c"},{2,"d"},{1,"e"},{3,"f"},{2,"g"}};
    show("original:       ", original);
    vector<Rec> v;
    v = original; insertionSort(v);              show("insertion sort: ", v);
    v = original; mergeSort(v);                  show("merge sort:     ", v);
    v = original; heapSort(v);                   show("heap sort:      ", v);
    v = original; quickSort(v, 0, (int)v.size()-1); show("quicksort:      ", v);
    cout << "\nRead the letters within each group of equal numbers. If they are in\n"
            "alphabetical order the sort was stable on this input.\n";
    return 0;
}
```

The output:

```
original:       3a 1b 3c 2d 1e 3f 2g
insertion sort: 1b 1e 2d 2g 3a 3c 3f
merge sort:     1b 1e 2d 2g 3a 3c 3f
heap sort:      1e 1b 2g 2d 3f 3c 3a
quicksort:      1b 1e 2d 2g 3c 3f 3a

Read the letters within each group of equal numbers. If they are in
alphabetical order the sort was stable on this input.
```

Read the tags. Insertion Sort and Merge Sort both give `1b 1e`, `2d 2g`, `3a 3c 3f` — every group in its original order. Heapsort gives `1e 1b`, `2g 2d`, `3f 3c 3a` — every group **reversed**. Quicksort gives `3c 3f 3a` — neither the original order nor the reverse, simply whatever fell out. That is the practical meaning of "unstable": not merely "may reorder", but "will reorder, unpredictably".

> **HOW TO GET STABILITY FROM AN UNSTABLE SORT.** Attach the original index to each record as a tie-breaker, and compare the pair (key,index) lexicographically. Now no two records have equal keys, so there is nothing to reorder, and any sort becomes stable. The cost is one extra integer per record and one extra comparison per tie. This is worth knowing: it means stability is never a reason you *cannot* use Quicksort, only a reason it costs a little more.

5.12.3 Cache behavior, and why Quicksort wins in practice

Here is the fact that **Table 5.7** cannot show, and it is the single most important practical point in this chapter.

The RAM model of Chapter 2 says every memory access costs one unit of time. Real hardware does not work that way. A modern processor reads memory in **cache lines** of sixty-four bytes, and keeps recently used lines in a small fast cache. Reading a value already in cache costs about one nanosecond; reading one that must come from main memory costs about a hundred. A factor of a hundred, invisible to every analysis in this chapter.

Now look at how our two $\Theta(n\lg n)$ in-place algorithms touch memory.

Quicksort scans. `PARTITION` walks from p to r, one element after another, in order. The processor's prefetcher recognizes the pattern immediately and fetches the next cache line before it is needed. Almost every access is a cache hit. Furthermore, as the recursion descends, the subarrays shrink, and once a subarray fits entirely in cache every remaining operation on it is free.

Heapsort jumps. `MAX-HEAPIFY` goes from index i to index $2i$ to index $4i$ — doubling each step. For a heap of a million elements, the walk from the root visits indices 1,2,5,11,23,47, … ,500000-ish, and the last few of those are in entirely different regions of memory, kilobytes or megabytes apart. Each such access is a cache miss. Worse, the pattern is unpredictable, so the prefetcher cannot help.

Listing 5.13 measures the difference on the same machine, with the same data.

Listing 5.13 A head-to-head benchmark of all four sorts on random data, with each sort given a fresh copy and its output checked.

```
// A head-to-head timing of the four sorts on the same data, on one machine.
#include <iostream>
#include <iomanip>
#include <vector>
#include <numeric>
#include <algorithm>
#include <random>
#include <chrono>
using namespace std;
using namespace std::chrono;

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

void mergeSortRec(vector<int>& A, vector<int>& buf, int p, int r) {
    if (p >= r) return;
    int q = p + (r - p) / 2;
    mergeSortRec(A, buf, p, q);
    mergeSortRec(A, buf, q + 1, r);
    int i = p, j = q + 1, k = p;
    while (i <= q && j <= r) buf[k++] = (A[i] <= A[j]) ? A[i++] : A[j++];
    while (i <= q) buf[k++] = A[i++];
    while (j <= r) buf[k++] = A[j++];
    for (int t = p; t <= r; ++t) A[t] = buf[t];
}
void mergeSort(vector<int>& A) { vector<int> buf(A.size()); mergeSortRec(A, buf, 0,
(int)A.size() - 1); }

void siftDown(vector<int>& A, int i, int m) {    // 0-based MAX-HEAPIFY
    while (true) {
        int l = 2 * i + 1, r = 2 * i + 2, largest = i;
        if (l < m && A[l] > A[largest]) largest = l;
        if (r < m && A[r] > A[largest]) largest = r;
        if (largest == i) return;
        swap(A[i], A[largest]);
        i = largest;
    }
}
void heapSort(vector<int>& A) {
    int m = (int)A.size();
    for (int i = m / 2 - 1; i >= 0; --i) siftDown(A, i, m);
```

Listing 5.13 (Continued) *A head-to-head benchmark of all four sorts on random data, with each sort given a fresh copy and its output checked.*

```
    for (int i = m - 1; i >= 1; --i) { swap(A[0], A[i]); siftDown(A, 0, i); }
}

mt19937 rng(20260919);
void quickSortRec(vector<int>& A, int p, int r) {
    while (p < r) {
        uniform_int_distribution<int> pick(p, r);
        swap(A[r], A[pick(rng)]);
        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]);
        int q = i + 1;
        if (q - p < r - q) { quickSortRec(A, p, q - 1); p = q + 1; }
        else               { quickSortRec(A, q + 1, r); r = q - 1; }
    }
}
void quickSort(vector<int>& A) { quickSortRec(A, 0, (int)A.size() - 1); }

template <class F>
double timeIt(F f, vector<int> data) {           // the copy is deliberate: each sort gets
fresh data
    auto t0 = high_resolution_clock::now();
    f(data);
    auto t1 = high_resolution_clock::now();
    for (size_t i = 1; i < data.size(); ++i) if (data[i - 1] > data[i]) return -1;  //
sanity check
    return duration<double, milli>(t1 - t0).count();
}

int main() {
    cout << fixed << setprecision(2);
    cout << "  n       | insertion |   merge |    heap |   quick   (milliseconds)\n";
    for (int n : {1000, 10000, 100000, 1000000}) {
        vector<int> data(n);
        for (int& x : data) x = (int)(rng() % 1000000000u);
        double ti = (n <= 100000) ? timeIt(insertionSort, data) : -2;
        double tm = timeIt(mergeSort, data);
        double th = timeIt(heapSort,  data);
        double tq = timeIt(quickSort, data);
        cout << setw(8) << n << " | ";
        if (ti < -1) cout << setw(9) << "skipped"; else cout << setw(9) << ti;
        cout << " | " << setw(7) << tm << " | " << setw(7) << th << " | " << setw(7) << tq
<< "\n";
    }
    return 0;
}
```

This is a *timing* listing, so its numbers are wall-clock milliseconds and will differ on every machine and between runs; what is reproducible is the pattern, not the digits. One run, on the machine this book was built on, produced:

```
  n       | insertion |   merge |    heap |   quick   (milliseconds)
    1000 |      0.09 |    0.05 |    0.03 |    0.04
   10000 |      7.82 |    0.59 |    0.37 |    0.48
  100000 |    806.91 |    7.48 |    6.41 |    6.93
 1000000 |   skipped |   97.02 |  122.72 |   66.79
```

Absolute times vary between machines, but the *shape* of this table does not, and it tells three stories.

Story one: quadratic really is quadratic. Insertion Sort goes $0.09 \to 7.82 \to 806.91$ milliseconds. Each tenfold increase in n multiplies the time by about a hundred, which is 10^2 —

exactly what $\Theta(n^2)$ predicts. At a million elements it was skipped because it would have taken well over a minute. The $n\lg n$ sorts went from 7.48 to 97.02 over the same step: a factor of thirteen, and the prediction is $10 \times \lg(10^6)/\lg(10^5) = 10 \times 1.2 = 12$. Exactly what $\Theta(n\lg n)$ predicts.

Story two: Heapsort wins while the data fits in cache, and loses when it does not. At $n = 100{,}000$ the array is four hundred kilobytes, which fits comfortably in a modern L2 cache, and Heapsort is the fastest of the three at 6.41 ms against Quicksort's 6.93. It wins there even though it makes more comparisons than Quicksort (roughly $2n\lg n$ against about $1.39\, n\lg n$), because its whole array sits in cache and it moves less data than Merge Sort. At $n = 1{,}000{,}000$ the array is four megabytes, the heap's jumps start missing cache, and Heapsort becomes the **slowest** of the three at 122.72 ms while Quicksort is fastest at 66.79 ms. Nothing in the asymptotic analysis changed. The hardware did.

Story three: this is why libraries choose Quicksort. Heapsort has the better guarantee and the worse constant. On the data sizes that matter — large ones — the constant wins: here Heapsort takes $122.72/66.79 \approx 1.84$ times as long as Quicksort on a million elements. A library that shipped Heapsort would be nearly twice as slow on every large sort, in exchange for protection against a case that randomization already makes vanishingly rare. Introsort (Section 5.10.3) takes both: Quicksort's speed by default, Heapsort's guarantee as a fallback.

5.12.4 When to use what

Table 5.8 is the decision table. Read it as advice, not law: measure your own data.

Table 5.8 Choosing a sorting algorithm. The last row is the one you will use most often in practice.

Situation	Best choice, and why
Small array, $n < 50$	**Insertion Sort** — the lowest overhead per element; the asymptotics do not matter yet
Nearly sorted data	**Insertion Sort** — $\Theta(n)$ when each element is close to its final place
General purpose, speed first	**Randomized Quicksort** (or the library's Introsort) — fastest in practice, thanks to cache behavior
A hard $O(n\lg n)$ guarantee is required	**Heapsort** or **Merge Sort** — neither has an n^2 case at all
Stability is required	**Merge Sort** — the only stable $\Theta(n\lg n)$ sort we have met
Memory is very tight	**Heapsort** — $O(1)$ extra space with a guaranteed $\Theta(n\lg n)$
Data larger than memory	**Merge Sort** — it is the basis of every external, multi-pass sort
Repeated insert and extract-max	**Binary heap / priority queue** — do not sort at all; $O(\lg n)$ per operation
Real-time system, hard deadline	**Heapsort** — a worst case you can put in a specification
Competitive programming	The language's built-in sort: `std::sort`, `Arrays.sort`, `sorted()`

THE ONE-SENTENCE SUMMARY. Use the library's sort unless you have a specific reason not to; if you need a worst-case guarantee, use Heapsort; if you need stability, use Merge Sort; if the array is tiny or almost sorted, use Insertion Sort; and if you are repeatedly asking for the largest remaining item, do not sort at all — use a heap.

5.13 Profitina in Practice

5.13.1 Where this chapter appears in a real product

Profitina is an AI-visibility platform — a *Generative Engine Optimization* (GEO) tool. It scans a brand's website, scores how ready each page is to be **cited by an AI answer engine**, and explains what to fix. The public product runs at `app.profitina.com`; it is a Python web application on PostgreSQL, using Ollama with a locally hosted Qwen2.5 model for its content and technical assistant, so the core loop does not depend on any hosted model [13].

Three parts of that product are this chapter, almost line for line.

The crawl queue is a priority queue. A scan of a site is not "fetch every page in the order we found them". Some pages matter far more than others: the home page, the pricing page, the pages an answer engine is most likely to be asked about. Others matter less: a paginated archive from four years ago. And the crawler has a budget — a per-tier limit on how many pages it will fetch in one scan. So the crawler keeps its frontier in a **max-priority queue** keyed by an urgency score, and repeatedly performs `EXTRACT-MAX` until the budget runs out. When a page turns out to link to something important, the linked page's urgency is raised — that is `INCREASE-KEY`. When a new URL is discovered, that is `INSERT`. All three are $O(\lg n)$ on a heap, and all three would be $O(n)$ on any array-based alternative. With a frontier of a hundred thousand URLs, $\lg n \approx 17$; the alternative is 100,000. The data structure choice is the difference between a scan that finishes and one that times out.

The re-scan scheduler is a min-priority queue. Every brand on a paid tier is re-scanned automatically at a cadence set by its tier. That is a classic event queue: each brand carries a "next due" timestamp, and the scheduler repeatedly extracts the *earliest* due item, runs the scan, recomputes the next due time, and re-inserts. This is the discrete-event simulation pattern of Section 5.7.2, running for real. A min-heap keeps the "what is due next?" question at $O(1)$ and the reschedule at $O(\lg n)$, no matter how many brands are enrolled.

The historical dashboard sorts. The weekly score-trend view, and the CSV export that goes with it, both sort rows — by date, by score, by signal. These are ordinary sorts over a few thousand rows, and the right answer is to let PostgreSQL or the standard library do it. The reason to understand this chapter is not to re-implement them; it is to know that sorting a thousand rows costs microseconds and sorting ten million costs seconds, so that you can tell which query needs an index and which does not.

What this section describes is the *shape* of the problem — a bounded-budget prioritized traversal, a periodic scheduler, a table that needs ordering — and the class of solution, which is the material of this chapter. That is the part that transfers to any system you build [13].

One more constraint worth naming, because it shapes design more than performance does. Profitina operates under Indonesia's personal-data law, **UU PDP 27/2022**. Any queue, log or export that touches personal data must be built so that a user's data can be found, exported and purged on request. In practice that means: key your queues on identifiers, not on personal fields; never put a name, an email or a phone number into a sort key or a cache key where it will be copied into places you cannot enumerate. The case study below follows that rule.

5.13.2 The worked case study: Profitina Laundry

Our hands-on case study is **Profitina Laundry**, the teaching business we use throughout this course. It is a real-sized problem.

The shop takes in washing over the course of a day. Each order has a weight in kilograms, a promised ready-by hour, and possibly an express surcharge. There are fewer machines than orders, so the shop must constantly answer one question: **what do we wash next?**

That is exactly `EXTRACT-MAX` on a priority queue. We define an urgency score that combines the three facts we know:

$$\text{urgency}(o) = (48 - \text{dueHour}) \times 10 + 40 \cdot [\, o \text{ is express} \,] - \text{kg}. \tag{5.19}$$

Read the three terms of Equation (5.19):

- $(48 - \text{dueHour}) \times 10$ — the earlier an order is due, the larger this is. The multiplier of 10 makes each hour worth ten points, so time dominates.
- $+40$ for an express order — worth four hours' head start, which is the shop's commercial policy expressed as a number.
- $-\text{kg}$ — a small penalty for heavy loads, so that when two orders are otherwise equal the lighter one goes first and clears the machine sooner.

At the end of the day the shop wants a takings report ordered by value, which is an ordinary sort — and we use Quicksort for it, since nothing here is adversarial and n is small.

Listing 5.14 is the whole thing, and it runs. Note what is *not* in the `Order` record: no customer name, no phone number, no address. Orders are identified by an anonymous ticket code printed on the customer's slip. This is not decoration; it is what UU PDP 27/2022 compliance looks like at the level of a single `struct` [13].

Listing 5.14 Profitina Laundry: a heap decides the washing order, and Quicksort produces the takings report.

```
// Profitina Laundry: the day's work queue as a heap, and the day's report as a quicksort.
// No personal data appears anywhere: an order is identified only by its ticket code.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

struct Order {
    string ticket;      // anonymous ticket code printed on the customer's slip
    int    kg;          // weight of the load
    int    dueHour;     // hour of the day the load was promised for, 24-hour clock
    bool   express;     // express orders were paid for at the higher rate
};

// Urgency: an express order is worth a four-hour head start, and every kilogram
// that a machine will be busy with pushes the order slightly later in the queue.
int urgency(const Order& o) {
    return (48 - o.dueHour) * 10 + (o.express ? 40 : 0) - o.kg;
}

// ---------- a max-heap of orders, keyed by urgency ----------
vector<Order> H;                                    // H[0] unused; the heap is H[1..heapSize]
int heapSize = 0;

void maxHeapify(int i) {
    int l = 2 * i, r = 2 * i + 1, largest = i;
    if (l <= heapSize && urgency(H[l]) > urgency(H[largest])) largest = l;
    if (r <= heapSize && urgency(H[r]) > urgency(H[largest])) largest = r;
    if (largest != i) { swap(H[i], H[largest]); maxHeapify(largest); }
}

void insertOrder(const Order& o) {
    ++heapSize;
    if ((int)H.size() <= heapSize) H.push_back(o); else H[heapSize] = o;
    int i = heapSize;
    while (i > 1 && urgency(H[i / 2]) < urgency(H[i])) { swap(H[i], H[i / 2]); i /= 2; }
}
```

Listing 5.14 (Continued) *Profitina Laundry: a heap decides the washing order, and Quicksort produces the takings report.*

```
Order extractMax() {
    Order best = H[1];
    H[1] = H[heapSize];
    --heapSize;
    maxHeapify(1);
    return best;
}

// ---------- quicksort, used for the end-of-day report ----------
int priceOf(const Order& o) { return o.kg * (o.express ? 12000 : 7000); }   // rupiah

int partition(vector<Order>& A, int p, int r) {
    int x = priceOf(A[r]), i = p - 1;
    for (int j = p; j <= r - 1; ++j) if (priceOf(A[j]) <= x) swap(A[++i], A[j]);
    swap(A[i + 1], A[r]);
    return i + 1;
}
void quicksortByPrice(vector<Order>& A, int p, int r) {
    if (p >= r) return;
    int q = partition(A, p, r);
    quicksortByPrice(A, p, q - 1);
    quicksortByPrice(A, q + 1, r);
}

int main() {
    vector<Order> arrivals = {
        {"PL-0341", 6, 17, false}, {"PL-0342",  3, 14, true },
        {"PL-0343", 12, 20, false}, {"PL-0344", 4, 13, false},
        {"PL-0345",  8, 16, true }, {"PL-0346", 2, 22, false},
        {"PL-0347",  5, 15, false}, {"PL-0348", 9, 18, true }
    };

    H.assign(1, arrivals[0]);                    // slot 0 is a placeholder, never read

    cout << "Orders as they arrive at the counter:\n";
    cout << "  ticket    kg  due  express  urgency\n";
    for (const Order& o : arrivals) {
        cout << "  " << o.ticket << setw(5) << o.kg << setw(5) << o.dueHour
             << setw(9) << (o.express ? "yes" : "no") << setw(9) << urgency(o) << "\n";
        insertOrder(o);
    }
```

Listing 5.14 (Continued) *Profitina Laundry: a heap decides the washing order, and Quicksort produces the takings report.*

```
    cout << "\nThe machine asks the queue what to wash next:\n";
    vector<Order> done;
    int step = 1;
    while (heapSize > 0) {
        Order o = extractMax();
        cout << "  wash " << step++ << ": " << o.ticket
             << "  (urgency " << urgency(o) << ", due " << o.dueHour << ":00"
             << (o.express ? ", express" : "") << ")\n";
        done.push_back(o);
    }

    cout << "\nEnd-of-day report, sorted by value with quicksort:\n";
    quicksortByPrice(done, 0, (int)done.size() - 1);
    long long total = 0;
    for (const Order& o : done) {
        cout << "  " << o.ticket << setw(4) << o.kg << " kg"
             << setw(12) << priceOf(o) << " IDR\n";
        total += priceOf(o);
    }
    cout << "  " << string(7, ' ') << setw(12) << " " << "------------\n";
    cout << "  takings" << setw(16) << total << " IDR\n";
    return 0;
}
```

The output:

```
Orders as they arrive at the counter:
  ticket    kg  due  express  urgency
  PL-0341    6   17       no      304
  PL-0342    3   14      yes      377
  PL-0343   12   20       no      268
  PL-0344    4   13       no      346
  PL-0345    8   16      yes      352
  PL-0346    2   22       no      258
  PL-0347    5   15       no      325
  PL-0348    9   18      yes      331

The machine asks the queue what to wash next:
  wash 1: PL-0342  (urgency 377, due 14:00, express)
  wash 2: PL-0345  (urgency 352, due 16:00, express)
  wash 3: PL-0344  (urgency 346, due 13:00)
  wash 4: PL-0348  (urgency 331, due 18:00, express)
  wash 5: PL-0347  (urgency 325, due 15:00)
  wash 6: PL-0341  (urgency 304, due 17:00)
  wash 7: PL-0343  (urgency 268, due 20:00)
  wash 8: PL-0346  (urgency 258, due 22:00)

End-of-day report, sorted by value with quicksort:
  PL-0346   2 kg       14000 IDR
  PL-0344   4 kg       28000 IDR
  PL-0347   5 kg       35000 IDR
  PL-0342   3 kg       36000 IDR
  PL-0341   6 kg       42000 IDR
  PL-0343  12 kg       84000 IDR
  PL-0345   8 kg       96000 IDR
  PL-0348   9 kg      108000 IDR
                     ------------
  takings          443000 IDR
```

Look at the wash order and check it against the scoring rule, because this is where the arithmetic becomes a business decision.

Order `PL-0344` is due at 13:00 — **earlier than anything else in the shop** — and yet it is washed *third*. Two express orders due at 14:00 and 16:00 go ahead of it. That is not a bug; it is the policy the score encodes. Express is worth 40 points, and one hour is worth 10, so an express order beats an ordinary one due up to four hours earlier. `PL-0342` is express and due at 14:00: score $(48-14)\times 10+40-3=340+40-3=377$. `PL-0344` is ordinary and due at 13:00: $(48-13)\times 10+0-4=350-4=346$. So $377>346$ and the express order wins.

> **THE LESSON FOR REAL SYSTEMS.** The heap does not decide anything; the *scoring function* does. The heap merely implements the decision efficiently. If the shop owner decides that a missed deadline is worse than a disappointed express customer, she changes the multiplier from 40 to 15, and the queue behaves differently the same afternoon — no change to the data structure at all. **Get the priority function right first; then reach for the heap to make it fast.** Section 5.7's four operations are the mechanism; the score is the policy. Profitina's own ranking works the same way: the scanner's job is to measure signals, and the scoring function's job is to say what they are worth. The app advises; the user decides.

5.14 Common Mistakes, and How to Avoid Them

1. Believing a heap is sorted. It is not. $A[1]$ is the maximum, and that is the only global guarantee. Printing a heap and expecting ascending order is the commonest beginner error. *Fix:* if you want sorted output, run Heapsort, or call `EXTRACT-MAX` repeatedly.

2. Mixing up `heap-size` and the array length. These are different numbers, and Heapsort depends on their being different. If `MAX-HEAPIFY` uses the array length instead of `heap-size`, it will happily pull already-sorted elements back into the heap and your sort will produce garbage. *Fix:* make `heap-size` an explicit variable, and never write `A.size()` inside `MAX-HEAPIFY`.

3. Bounds-checking after the array access. Writing `if (A[l] > A[largest] && l <= heapSize)` reads out of range before testing. C++ evaluates left to right and `&&` short-circuits, so the bounds test must come **first**. *Fix:* `if (l <= heapSize && A[l] > A[largest])`, always in that order.

4. Comparing the right child against `A[i]` instead of `A[largest]`. After the left-child test may have updated `largest`, the right-child test must compare against the current champion. *Fix:* the second `if` always reads `A[r] > A[largest]`.

5. Building a heap by inserting n times. It works, and it costs $O(n\lg n)$. `BUILD-MAX-HEAP` costs $O(n)$. *Fix:* if you have all the data up front, build; insert only when items genuinely arrive one at a time. In C++, `make_heap` is the $O(n)$ version.

6. Starting `BUILD-MAX-HEAP` at n instead of $\lfloor n/2\rfloor$. Harmless but wasteful — the second half of the array is leaves, and calling `MAX-HEAPIFY` on a leaf does nothing. *Fix:* start at $\lfloor n/2\rfloor$. Starting at 1 and counting *up* is the serious version of this error: the precondition fails and the result is wrong, not merely slow.

7. Getting the Quicksort recursion bounds wrong. With Lomuto's partition, the calls are `(p, q-1)` and `(q+1, r)`. With Hoare's, they are `(p, q)` and `(q+1, r)` — and the pivot value must then be taken from $A[p]$, not $A[r]$ (Section 5.8.5). Mixing the two gives either an infinite recursion or a lost element. *Fix:* write the partition and its two recursive calls at the same time, never separately.

8. Deploying a fixed-pivot Quicksort. Sorted input then triggers the $\Theta(n^2)$ worst case, and sorted input is extremely common. *Fix:* randomize the pivot — two lines — or use median-of-three, or use the library.

9. Forgetting that the recursion consumes stack. Naive Quicksort on a sorted million-element array recurses a million deep and crashes before the quadratic time becomes visible. *Fix:* always recurse on the smaller side and loop on the larger, as in Section 5.10.4.

10. Assuming randomization removes the worst case. It does not; it makes it improbable. If you need a *guarantee* — a real-time deadline, an adversarial input, a service-level agreement — use Heapsort or Introsort, not randomized Quicksort. *Fix:* be clear whether you need "fast on average" or "never slow". They are different requirements.

11. Assuming Quicksort or Heapsort is stable. Neither is. If two records with equal keys must keep their order, either use Merge Sort or add the original index as a tie-breaker. *Fix:* see the note at the end of Section 5.12.2.

12. Judging by asymptotics alone. **Table 5.7** says Heapsort and Quicksort have the same average complexity. The benchmark in Section 5.12.3 shows Quicksort finishing a million elements in 67 ms and Heapsort in 123 ms. The tables do not measure cache. *Fix:* measure, on your data, on your machine.

13. Sorting when you should be using a heap. If the task is "repeatedly take the largest remaining item", do not sort the array — that is $\Theta(n\lg n)$ up front and you may only need the first few items. A heap gives you the first k in $O(n + k\lg n)$. *Fix:* ask whether you need the whole order or just the extremes.

14. Using `int` where the count can overflow. A comparison count of $n(n-1)/2$ for $n = 100{,}000$ is about 5×10^9, which overflows a 32-bit `int`. *Fix:* count with `long long`, as every program in this chapter does.

15. Writing your own sort in a contest. `std::sort` is Introsort, is correct, and is faster than what you will write. *Fix:* use it, and spend the time you saved on the part of the problem that is actually hard.

5.15 Chapter Summary

1. **A binary heap stores a nearly complete binary tree inside a flat array**, with no pointers at all. Navigation is pure arithmetic: $\text{Parent}(i) = \lfloor i/2 \rfloor$, $\text{Left}(i) = 2i$, $\text{Right}(i) = 2i + 1$, each a single shift instruction. Because the array is contiguous, heaps are cache-friendly in a way that pointer-based trees are not.
2. **The max-heap property is local**: every parent is at least as large as its children, and nothing more is guaranteed. From that one local rule the single global fact follows by transitivity: $A[1]$ is the maximum.
3. **A heap of n elements has height exactly** $\lfloor \lg n \rfloor$, its leaves occupy indices $\lfloor n/2 \rfloor + 1$ through n, and at most $\lceil n/2^{h+1} \rceil$ of its nodes have height h. Every cost in the chapter is derived from those three counts.
4. **`MAX-HEAPIFY` sinks one value down one root-to-leaf path** in $O(\lg n)$, and it is the engine of everything else. Its recurrence Equation (5.5), $T(n) \le T(2n/3) + \Theta(1)$, solves to $O(\lg n)$ by Case 2 of the Master Theorem.
5. **`BUILD-MAX-HEAP` runs in $\Theta(n)$, not $\Theta(n\lg n)$** — proved by summing $\lceil n/2^{h+1} \rceil \cdot O(h)$ over all heights and using $\sum_{h \ge 0} h/2^h = 2$ (Equation (5.8)), which itself comes from differentiating the

geometric series. Measurement confirms it: the number of exchanges per element stays near 0.74 from a thousand elements to ten million, while $\lg n$ climbs from 9 to 23.

6. **Heapsort is `BUILD-MAX-HEAP` plus $n-1$ extractions**: $\Theta(n\lg n)$ in every case when the keys are distinct, $O(1)$ extra space, not stable. It is the algorithm that fills the hole in **Table 5.1**.

7. **A priority queue on a heap does `MAXIMUM` in $O(1)$ and `INSERT`, `EXTRACT-MAX` and `INCREASE-KEY` in $O(\lg n)$** — the only implementation in **Table 5.4** with no linear operation. This is the data structure behind Dijkstra, Prim, Huffman, event simulation and every scheduler you will ever write.

8. **Quicksort partitions around a pivot and recurses**, with no combine step. Lomuto's partition maintains a four-region invariant and is easy to prove; Hoare's runs two pointers toward each other, does fewer exchanges, and is what real libraries use.

9. **Quicksort's cost is entirely a function of how balanced the splits are.** Perfect splits give $\Theta(n\lg n)$; the degenerate $1:(n-1)$ split gives $\Theta(n^2)$; and **any split in a fixed constant ratio — even** $99:1$ **— gives** $\Theta(n\lg n)$, differing only in the constant. That robustness is why Quicksort works.

10. **The expected number of comparisons is** $2(n+1)H_n - 4n \approx 1.39\, n\lg n$, proved by the indicator-variable argument: z_i and z_j are compared exactly when the first pivot chosen from Z_{ij} is one of them, which has probability $2/(j-i+1)$.

11. **Randomizing the pivot costs two lines and makes the running time independent of the input.** It does not eliminate the worst case; it reduces its probability to about $2^{n-1}/n!$ (Equation (5.16)), which for $n = 100$ is around 10^{-128}.

12. **Always recurse on the smaller side.** It bounds the stack depth at $\lg n$ regardless of how the partitions go, and it turns a crash into a non-event.

13. **No comparison sort can beat** $\Omega(n\lg n)$, because its decision tree needs $n!$ leaves and a binary tree with $n!$ leaves has height at least $\lg(n!) = \Omega(n\lg n)$ by Equation (5.18). Merge Sort, Heapsort and randomized Quicksort are therefore asymptotically optimal, and the remaining competition is over constants, memory and guarantees.

14. **In practice Quicksort beats Heapsort** despite the worse guarantee, because it scans memory sequentially while a heap jumps between indices i, $2i$ and $4i$. Measured: 67 ms against 123 ms on a million elements — while at a hundred thousand elements, where everything fits in cache, Heapsort is the faster of the two.

15. **Changing the data structure can change the complexity class.** Selection Sort and Heapsort are the same algorithm; the only difference is where the data lives. That is the most transferable idea in the chapter.

5.15.1 Looking ahead

Chapter 6 leaves arrays behind for **binary search trees** and their self-balancing form, **AVL trees**, which support search, insertion *and* deletion in $O(\lg n)$ — something a heap cannot do, since a heap can find only the maximum, not an arbitrary key. Chapter 7 moves from divide-and-conquer to **dynamic programming**, and Chapter 8, *Evaluation and Discussion II*, then works problems drawn from Chapters 5 to 7 — this chapter's heaps and partitions among them. Chapters 11 and 13 finally cash in the priority queue: Prim's minimum spanning tree and Dijkstra's shortest paths are both, structurally, Section 5.7 wrapped around a graph.

One topic deserves a pointer of its own, because this chapter proved it impossible and then quietly exempted it: **linear-time sorting**. Counting Sort, Radix Sort and Bucket Sort sort n items in $\Theta(n)$ by refusing to be comparison sorts at all. They lie outside this book's main line, and Chapter 8 of CLRS [3] treats them in full.

5.16 Exercises

5.1 What are the minimum and maximum numbers of elements in a heap of height h?

5.2 Show that an n-element heap has height $\lfloor \lg n \rfloor$ by computing the height directly for $n = 1,2,3,4,7,8,15,16,17$ and checking each against the formula.

5.3 Is the array $A = [23,17,14,6,13,10,1,5,7,12]$ a max-heap? If not, name every violation.

5.4 Where in a max-heap might the **smallest** element reside, assuming all elements are distinct? Prove your answer.

5.5 Illustrate the operation of `MAX-HEAPIFY(A, 3)` on $A = [27,17,3,16,13,10,1,5,7,12,4,8,9,0]$ with heap-size $= 14$. Show the array after every exchange.

5.6 Illustrate `BUILD-MAX-HEAP` on $A = [5,3,17,10,84,19,6,22,9]$. Show the array after each iteration of the loop.

5.7 Starting from the heap you built in Exercise 5.6, show the first three iterations of `HEAPSORT`.

5.8 What is the running time of `HEAPSORT` on an array A of length n that is **already sorted in increasing order**? What about **decreasing order**? Justify both answers.

5.9 Argue that for a heap with distinct elements, the maximum of the *leaves* alone need not be the second-largest element of the heap. Then state, and prove, where the second-largest element must be.

5.10 Illustrate `MAX-HEAP-INSERT(A, 10)` on the heap $A = [15,13,9,5,12,8,7,4,0,6,2,1]$. Show every comparison.

5.11 Using Lomuto's `PARTITION`, illustrate the operation on the array $A = [13,19,9,5,12,8,7,4,21,2,6,11]$. Show the value of i and j and the array at every iteration.

5.12 What value of q does `PARTITION` return when all elements of $A[p \ldots r]$ have the **same** value? Modify Lomuto's partition so that it splits such an array into two roughly equal halves, and say why that matters.

5.13 Solve the recurrence $T(n) = T(n-1) + \Theta(n)$ by the substitution method, proving $T(n) = O(n^2)$ with an explicit constant.

5.14 Suppose Quicksort's `PARTITION` always produces a $1:99$ split. Write the recurrence, draw the shape of its recursion tree, compute the minimum and maximum depth, and conclude the running time.

5.15 What is the probability that randomized Quicksort makes a *perfectly balanced* split at the top level of an array of n distinct elements? For $n = 1000$, how large is that probability?

5.16 Show, with a concrete four-element array, that Heapsort is not stable, and that Quicksort is not stable.

5.17 Prove that any comparison sort needs at least $\lceil \lg(n!) \rceil$ comparisons in the worst case, and compute that number for $n = 5$. Then design, by hand, a decision tree that sorts three elements in at most three comparisons.

5.18 *(Programming.)* Write a C++ program that reads n and k, generates n random integers, and prints the k largest using a heap in $O(n + k\lg n)$ time — without sorting the whole array. Compare its running time against `std::sort` for $n = 10^7$, $k = 10$.

5.19 *(Programming.)* Implement a min-priority queue supporting `INSERT`, `MINIMUM`, `EXTRACT-MIN` and `DECREASE-KEY`, and use it to merge k sorted lists of total length n in $O(n\lg k)$ time.

5.20 *(Analysis.)* The `BUILD-MAX-HEAP` measurement in **Listing 5.3** reported about 0.74 exchanges per element. Explain why that constant is less than 1, and estimate it from the heights of the nodes.

5.17 Solutions to the Exercises

5.1 By Equation (5.4) in Section 5.3.6, a heap of height h satisfies $2^h \le n < 2^{h+1}$, that is, $2^h \le n \le 2^{h+1} - 1$, since n is an integer. The **minimum** is 2^h: levels 0 through $h - 1$ full ($2^h - 1$ nodes) plus one node on level h. The **maximum** is $2^{h+1} - 1$: every level full, including level h. For example, height 3 admits between 8 and 15 elements.

5.2 Compute $\lfloor \lg n \rfloor$ and compare with the drawn tree; **Table 5.9** collects the nine cases.

Table 5.9 Heights for Exercise 5.2. The height increases by one exactly at each power of two, which is what the floor of a base-two logarithm does.

n	$\lg n$	$\lfloor \lg n \rfloor$	Actual height
1	0	0	0 ✓
2	1	1	1 ✓
3	1.58	1	1 ✓
4	2	2	2 ✓
7	2.81	2	2 ✓
8	3	3	3 ✓
15	3.91	3	3 ✓
16	4	4	4 ✓
17	4.09	4	4 ✓

5.3 Check every parent. $A[1] = 23 \ge 17,14$ ✓. $A[2] = 17 \ge 6,13$ ✓. $A[3] = 14 \ge 10,1$ ✓. $A[4] = 6$: children $A[8] = 5$ ✓ and $A[9] = 7$ — **violation**, $6 < 7$. $A[5] = 13$: child $A[10] = 12$ ✓. So there is exactly one violation, at the pair $(A[4], A[9]) = (6,7)$. The array is **not** a max-heap. One call to `MAX-HEAPIFY(A, 4)` repairs it, giving [23,17,14,7,13,10,1,5,6,12].

5.4 The smallest element must be a **leaf**, that is, at an index in $\lfloor n/2 \rfloor + 1, \ldots, n$. *Proof:* suppose the smallest element sat at an internal node i. Then i has at least one child c, and the heap property demands $A[i] \ge A[c]$. Since all elements are distinct, $A[i] > A[c]$, so $A[i]$ is not the smallest — a contradiction. Hence it is a leaf. Any leaf is possible: place the smallest value anywhere in the second half of a heap of otherwise-large values and the property still holds.

5.5 `MAX-HEAPIFY(A, 3)` on $A = [27,17,3,16,13,10,1,5,7,12,4,8,9,0]$, heap-size $= 14$.

- $i = 3$: $A[3] = 3$, children $A[6] = 10$, $A[7] = 1$. Largest is 10 at index 6. Swap: [27,17,**10**,16,13,**3**,1,5,7,12,4,8,9,0].
- $i = 6$: $A[6] = 3$, children $A[12] = 8$, $A[13] = 9$. Largest is 9 at index 13. Swap: [27,17,10,16,13,**9**,1,5,7,12,4,8,**3**,0].
- $i = 13$: $2 \times 13 = 26 > 14$, a leaf. Stop.

Final: $A = [27,17,10,16,13,9,1,5,7,12,4,8,3,0]$.

5.6 $A = [5,3,17,10,84,19,6,22,9]$, $n = 9$, $\lfloor n/2 \rfloor = 4$.

- $i = 4$: $A[4] = 10$, children $A[8] = 22$, $A[9] = 9$. Swap with 22: [5,3,17,**22**,84,19,6,**10**,9]. Index 8 is a leaf. Stop.
- $i = 3$: $A[3] = 17$, children $A[6] = 19$, $A[7] = 6$. Swap with 19: [5,3,**19**,22,84,**17**,6,10,9]. Leaf. Stop.
- $i = 2$: $A[2] = 3$, children $A[4] = 22$, $A[5] = 84$. Swap with 84: [5,**84**,19,22,**3**,17,6,10,9]. At index 5: $2 \times 5 = 10 > 9$, a leaf. Stop.
- $i = 1$: $A[1] = 5$, children $A[2] = 84$, $A[3] = 19$. Swap with 84: [**84**,**5**,19,22,3,17,6,10,9]. At index 2: children $A[4] = 22$, $A[5] = 3$; swap with 22: [84,**22**,19,**5**,3,17,6,10,9]. At index 4: children $A[8] = 10$, $A[9] = 9$; swap with 10: [84,22,19,**10**,3,17,6,**5**,9]. Index 8 is a leaf. Stop.

Result: $A = [84,22,19,10,3,17,6,5,9]$. Verify: $84 \geq 22,19$; $22 \geq 10,3$; $19 \geq 17,6$; $10 \geq 5,9$ ✓.

5.7 Starting from [84,22,19,10,3,17,6,5,9], $n = 9$.

- $i = 9$: swap $A[1] = 84$ with $A[9] = 9$; shrink to 8; heapify the root: 9 against 22 and 19 → swap with 22; then 9 against $A[4] = 10$ and $A[5] = 3$ → swap with 10; then index 4 has children $A[8] = 5$ only → $9 > 5$, stop. Result [22,10,19,9,3,17,6,5 | 84].
- $i = 8$: swap 22 with $A[8] = 5$; shrink to 7; heapify: 5 against 10 and 19 → swap with 19; at index 3, children $A[6] = 17$ and $A[7] = 6$ → swap with 17; index 6 is now a leaf. Result [19,10,17,9,3,5,6 | 22,84].
- $i = 7$: swap 19 with $A[7] = 6$; shrink to 6; heapify: 6 against 10 and 17 → swap with 17; at index 3, only child $A[6] = 5$, and $6 > 5$, stop. Result [17,10,6,9,3,5 | 19,22,84].

5.8 In **both** cases the running time is $\Theta(n\lg n)$. Heapsort's cost does not depend on the input order. `BUILD-MAX-HEAP` is $\Theta(n)$ either way, and the extraction loop performs $n - 1$ calls to `MAX-HEAPIFY`, each of which must sink a small element — taken from the bottom of the heap — through a heap of height $\Theta(\lg n)$. There is no input that shortens those walks systematically. This is Heapsort's virtue and its curse: it is never surprised, and it is never lucky. (Contrast Quicksort, for which the increasing-order input is the $\Theta(n^2)$ worst case.)

5.9 The maximum of the leaves need not be second-largest: in [16,14,10,8,7,9,3,2,4,1] the leaves are {9,3,2,4,1} with maximum 9, while the second-largest element of the heap is 14, an internal node. The correct statement is: **the second-largest element must be a child of the root**, that is, at index 2 or 3. *Proof:* let x be the second-largest element, and suppose it sat at depth 2 or more. Then x has a parent p, and p is not the root. The heap property gives $A[p] > x$ (strictly, since the elements are distinct), and the root, being the maximum, is also larger than x. So at least two different elements, the root and p, are larger than x, and x is at best the third largest — a contradiction. Hence x sits at depth 1, at index 2 or 3. ■

5.10 Insert 10 into $A = [15,13,9,5,12,8,7,4,0,6,2,1]$ ($n = 12$).

- heap-size ← 13; $A[13] \leftarrow -\infty$; then raise it to 10.

- $i = 13$. Parent(13) $= 6$. Is $A[6] = 8 < 10$? Yes $\rightarrow$ swap. Now the array reads $\ldots A[6] = 10 \ldots A[13] = 8$, and $i = 6$.
- $i = 6$. Parent(6) $= 3$. Is $A[3] = 9 < 10$? Yes $\rightarrow$ swap. Now $A[3] = 10$, $A[6] = 9$, and $i = 3$.
- $i = 3$. Parent(3) $= 1$. Is $A[1] = 15 < 10$? **No.** Stop.

Result: $A = [15,13,10,5,12,9,7,4,0,6,2,1,8]$. Three comparisons, two exchanges, and the new key rose two levels.

5.11 Lomuto on $A = [13,19,9,5,12,8,7,4,21,2,6,11]$, $p = 1$, $r = 12$, pivot $x = A[12] = 11$, $i = 0$. **Table 5.10** carries the loop.

Table 5.10 The Lomuto partition of Exercise 5.11, iteration by iteration.

j	*A[j]*	≤11?	*i* After	Array
1	13	no	0	[13,19,9,5,12,8,7,4,21,2,6,11]
2	19	no	0	unchanged
3	9	yes	1	[9,19,13,5,12,8,7,4,21,2,6,11]
4	5	yes	2	[9,5,13,19,12,8,7,4,21,2,6,11]
5	12	no	2	unchanged
6	8	yes	3	[9,5,8,19,12,13,7,4,21,2,6,11]
7	7	yes	4	[9,5,8,7,12,13,19,4,21,2,6,11]
8	4	yes	5	[9,5,8,7,4,13,19,12,21,2,6,11]
9	21	no	5	unchanged
10	2	yes	6	[9,5,8,7,4,2,19,12,21,13,6,11]
11	6	yes	7	[9,5,8,7,4,2,6,12,21,13,19,11]

Finally swap $A[i+1] = A[8]$ with $A[12]$: [9,5,8,7,4,2,6,**11**,21,13,19,12] and return $q = 8$. Left part {9,5,8,7,4,2,6}, all ≤ 11 ✓; right part {21,13,19,12}, all >11 ✓.

5.12 If every element equals the pivot then every test $A[j] \leq x$ succeeds, i increments on every iteration, and the procedure returns $q = r$ — the extreme split. Quicksort then recurses on $n-1$ elements and the cost is $\Theta(n^2)$: an array of a million identical values is a worst case. **Fix:** the smallest change to Lomuto's procedure itself is to share out the ties. Keep a flag that flips every time $A[j] = x$, and let an element equal to the pivot join the left region only when the flag is set. On an all-equal array, half of the $n-1$ scanned elements then go left and half stay right, so the split is into two halves of sizes $\lfloor (n-1)/2 \rfloor$ and $\lceil (n-1)/2 \rceil$; the sort is still correct, because an element equal to the pivot may sit on either side of it. A second remedy is Hoare's scheme, which stops both pointers on values equal to the pivot and therefore also splits an all-equal array almost exactly in half. The proper fix is the *Dutch national flag* partition: produce three regions $<x$, $=x$, $>x$ and recurse only on the outer two. An array with d distinct values then sorts in $O(n \lg d)$. This matters because duplicate-heavy data is extremely common — statuses, categories, dates, boolean flags.

5.13 Guess $T(n) \leq cn^2$ and let the $\Theta(n)$ term be at most dn.

Assume $T(n-1) \leq c(n-1)^2$. Then

$$T(n) \leq c(n-1)^2 + dn = cn^2 - 2cn + c + dn = cn^2 - (2c - d)n + c.$$

This is at most cn^2 provided $(2c-d)n \geq c$, that is, $n \geq \frac{c}{2c-d}$. Choosing $c = d$ makes the requirement $n \geq \frac{d}{d} = 1$, so the bound holds for all $n \geq 1$ with $c = d$ and the base case $T(1) \leq c$. Hence $T(n) = O(n^2)$. ■

5.14 The recurrence is $T(n) = T(n/100) + T(99n/100) + \Theta(n)$.

Every complete level costs n, since $\frac{1}{100}n + \frac{99}{100}n = n$. The shallowest leaf is reached by always taking the $1/100$ branch: depth $\log_{100} n$. The deepest is reached by always taking the $99/100$ branch: depth $\log_{100/99} n$. So

$$n\log_{100} n \leq T(n) \leq n\log_{100/99} n + \Theta(n).$$

Converting to base two: $\log_{100} n = \lg n/\lg 100 \approx 0.15\lg n$ and $\log_{100/99} n = \lg n/\lg(100/99) \approx \lg n/0.0145 \approx 69\lg n$. Both are $\Theta(n\lg n)$, so

$$T(n) = \Theta(n\lg n),$$

with a constant somewhere between about 0.15 and about 69 — a large constant, and still not a change of growth rate. This is the point of Section 5.9.4.

5.15 A perfectly balanced split requires the chosen pivot to be the **median**. With a uniformly random pivot out of n distinct elements, exactly one element is the median, so the probability is $1/n$. For $n = 1000$ that is 0.001, one chance in a thousand.

The point of the question is that this tiny probability does **not** matter. Quicksort does not need median pivots; by Section 5.9.4 it needs only a constant-ratio split. The probability that the pivot falls in the middle half of the sorted order — giving at worst a 1: 3 split — is $1/2$, and a 1: 3 split is already $\Theta(n\lg n)$ with depth $\log_{4/3} n \approx 2.4\lg n$. So half of all pivots are "good enough", which is why Quicksort behaves well essentially always.

5.16 Take the records $\langle 2, a\rangle, \langle 1, b\rangle, \langle 2, c\rangle, \langle 1, d\rangle$, sorted by the number.

Heapsort. `BUILD-MAX-HEAP` on $[2a, 1b, 2c, 1d]$: at $i = 2$, $A[2] = 1b$ is compared with its child $A[4] = 1d$ — equal, so no swap (the test is a strict $>$); at $i = 1$, $2a$ is compared with $1b$ and $2c$ — it is not beaten, so no swap. The heap is unchanged. Now the extraction loop, one iteration at a time:

- $i = 4$: swap $A[1] = 2a$ with $A[4] = 1d$, giving $[1d, 1b, 2c \mid 2a]$. Heapify the root: $2c > 1d$, so swap, giving $[2c, 1b, 1d \mid 2a]$.
- $i = 3$: swap $A[1] = 2c$ with $A[3] = 1d$, giving $[1d, 1b \mid 2c, 2a]$. Heapify: $1b$ is not greater than $1d$, so nothing moves.
- $i = 2$: swap $A[1] = 1d$ with $A[2] = 1b$, giving $[1b \mid 1d, 2c, 2a]$.

The result is $[1b, 1d, 2c, 2a]$. The two 2s now appear in the order c, a, the reverse of their original order. **Unstable.**

Quicksort (Lomuto, pivot last). $x = 1d$. $j = 1$: $2a > 1$, skip. $j = 2$: $1b \leq 1$, $i \to 1$, swap $A[1] \leftrightarrow A[2] \to [1b, 2a, 2c, 1d]$. $j = 3$: $2c > 1$, skip. Then swap $A[2] \leftrightarrow A[4] \to [1b, 1d, 2c, 2a]$. The two 2s are now in the order c, a — reversed. **Unstable.**

The general reason is the same in both cases: both algorithms move elements across long distances in a single exchange, past elements they never compared with.

5.17 *The bound.* By Section 5.11.3, a correct comparison sort's decision tree has at least $n!$ leaves and a binary tree of height h has at most 2^h leaves, so $2^h \geq n!$, hence $h \geq \lg(n!)$. Because h counts comparisons and is an integer, $h \geq \lceil \lg(n!) \rceil$.

For $n = 5$: $5! = 120$ and $\lg 120 \approx 6.91$, so at least $\lceil 6.91 \rceil = \mathbf{7}$ comparisons are needed. (Seven are also sufficient — merge-insertion sort achieves it — so the bound is tight here.)

A decision tree for three elements a, b, c. Three comparisons suffice, and by the bound $\lceil \lg 6 \rceil = 3$ are necessary.

```
                    a : b
            ≤ /             \ >
          b : c             a : c
      ≤ /       \ >     ≤ /       \ >
  (a,b,c)     a : c   (b,a,c)    b : c
            ≤ /   \ >         ≤ /    \ >
        (a,c,b) (c,a,b)    (b,c,a)  (c,b,a)
```

Six leaves, one per permutation, and no root-to-leaf path is longer than three. ✓

5.18 *Sketch of the solution.* Two approaches: the first is exactly what the question asks for, and the second is a useful alternative.

Approach A ($O(n + k\lg n)$). Build a max-heap of all n elements with `make_heap` — $O(n)$ — then call `pop_heap` k times, $O(k\lg n)$. Total $O(n + k\lg n)$, which is what the question wants.

Approach B ($O(n\lg k)$, better when $k \ll n$). Keep a **min**-heap of size exactly k. For each of the n elements: if the heap has fewer than k items, push it; otherwise, if the element exceeds the heap's minimum, pop the minimum and push the element. At the end the heap holds the k largest. Cost $O(n\lg k)$, and — crucially — $O(k)$ memory rather than $O(n)$, so it works on a stream you cannot store.

In C++, Approach A is `make_heap(v.begin(), v.end()); for (int t=0;t<k;++t){ pop_heap(v.begin(), v.end()-t); }`, and Approach B is a `priority_queue<int, vector<int>, greater<int>>` of capacity k. Both will finish $n = 10^7$, $k = 10$ in well under a second, while `std::sort` on 10^7 integers takes roughly a second — because sorting computes the full order when only ten values were wanted. The lesson is Common Mistake 13.

5.19 *Sketch.* Put one entry per list into a min-heap, each entry recording `(value, whichList, positionInList)`. Repeatedly `EXTRACT-MIN`, append its value to the output, and `INSERT` the next element of that same list, if any. The heap never holds more than k entries, so each of the n extractions and n insertions costs $O(\lg k)$: total $O(n\lg k)$.

This is strictly better than concatenating and sorting, which costs $O(n\lg n)$, and it is how external sorting merges runs from disk. `DECREASE-KEY` is not needed for the merge itself, but is required in the same class's other famous client, Dijkstra's algorithm (Chapter 13). The `DECREASE-KEY` implementation is the exact mirror of `INCREASE-KEY` in **Listing 5.5**: assign the smaller key, then float the element *upward* while its parent is larger.

5.20 The exchange count is the total distance all values sink during `BUILD-MAX-HEAP`. Group the nodes by height:

- Height 0 — about $n/2$ nodes, the leaves. `MAX-HEAPIFY` is never called on them. **Zero** exchanges.
- Height 1 — about $n/4$ nodes. Each sinks at most one level, and on random data sinks about $\frac{2}{3}$ of a level on average (the root of a three-element subtree is already the largest with probability $1/3$). Contribution $\approx \frac{n}{4} \cdot \frac{2}{3} \approx 0.167n$.

- Height 2 — about $n/8$ nodes, sinking at most two levels; on random data a little over one level. Contribution $\approx \frac{n}{8} \cdot 1.2 \approx 0.15n$.
- Height h — about $n/2^{h+1}$ nodes sinking $O(h)$ levels.

Summing gives $n \sum_{h \geq 0} c_h / 2^{h+1}$ with $c_h < h$, and the series converges to a constant strictly less than $\sum_h h / 2^{h+1} = 1$. The measured 0.74 sits comfortably under that ceiling — and the ceiling itself is exactly half of the $\sum h/2^h = 2$ identity of Section 5.5.6. The reason the constant is below 1 is the first bullet: **more than half of all nodes are never touched at all.**

References

[1] J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, Jun. 1964, doi: 10.1145/512274.512284.

[2] R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, vol. 7, no. 12, p. 701, Dec. 1964, doi: 10.1145/355588.365103.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, chs. 6–8.

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

[5] ISO/IEC 14882:2024, *Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2024.

[6] Python Software Foundation, "heapq — Heap queue algorithm," *The Python Standard Library*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/heapq.html. [Accessed: 24-Sep-2026].

[7] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, Jan. 1962, doi: 10.1093/comjnl/5.1.10.

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

[9] Oracle, "Class Arrays," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Arrays.html. [Accessed: 24-Sep-2026].

[10] D. R. Musser, "Introspective sorting and selection algorithms," *Software: Practice and Experience*, vol. 27, no. 8, pp. 983–993, Aug. 1997, doi: 10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-#.

[11] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011, ch. 2.

[12] S. Wild, "Powersort in official Python 3.11 release." [Online]. Available: https://www.wild-inter.net/posts/powersort-in-python-3.11. [Accessed: 24-Sep-2026]. See also J. I. Munro and S. Wild, "Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs," in *Proc. 26th European Symposium on Algorithms (ESA 2018)*, LIPIcs vol. 112, 2018, pp. 63:1–63:16.

[13] Profitina, "Profitina — AI Visibility (GEO) platform." [Online]. Available: https://profitina.com. [Accessed: 19-Sep-2026].

Chapter 6

Binary Search Trees and AVL Trees

By the end of this chapter you will be able to build, by hand and in C++, a data structure that keeps a collection of keys permanently in order while letting you add and remove keys at will. You will be able to state the binary-search-tree property precisely, and say exactly how it differs from the heap property that so many newcomers confuse it with. You will be able to walk a tree three different ways, and prove — not assert — that one of those three walks prints the keys in sorted order. You will be able to search a tree, find its smallest and largest key, find the key that comes immediately before or immediately after any given key, insert a key, and delete a key in all three of the shapes deletion can take, and you will know why every one of those operations costs $O(h)$ where h is the height. You will then see how h can quietly grow to n and destroy everything, and you will learn Adelson-Velsky and Landis's cure: a tree that repairs its own shape after every change. You will derive, from first principles, why an AVL tree can never be taller than about $1.44\lg n$; you will master all four rotation cases with the exact pointer assignments each one performs; you will replay a five-step insertion example and a deletion example node by node; and you will finish able to say, for a given job, whether an AVL tree, a red-black tree or a hash table is the right tool.

6.1 Where We Left Off

Chapter 3 gave us a language and a toolbox.

The language was **asymptotic notation**. We learned that $O(g(n))$ is the set of functions that eventually sit below some constant multiple of g; that $\Omega(g(n))$ is the set that eventually sits above some constant multiple of g; and that $\Theta(g(n))$ is the set that does both at once, trapped in a corridor between $c_1 g(n)$ and $c_2 g(n)$. We learned to name the witnesses c and n_0 rather than wave at them.

The toolbox was **recurrences**. When an algorithm calls itself, its running time obeys an equation that mentions itself, and we learned three ways to solve such an equation: substitution (guess and prove by induction), the recursion tree (draw the work level by level and add it up), and the Master Theorem (a recipe that answers $T(n) = aT(n/b) + f(n)$ in three cases).

Those tools were built to analyze algorithms. Now we are going to use them to analyze a *data structure*, and that is a slightly different exercise. An algorithm has one running time. A data structure has a running time per operation, and the running times of its operations are tangled together: the shape that makes one operation fast is produced by the other operations. That entanglement is the subject of this chapter.

Here is the gap we are trying to close. In Chapter 2 we met two ways of holding a collection of keys.

A **sorted array** is wonderful to search. Binary search halves the candidate range at every probe, so it finds a key among n of them in $\Theta(\lg n)$ probes. It is miserable to modify: to insert a new key in its proper place you must shift, on average, half the array one position to the right, and that is $\Theta(n)$ work per insertion.

A **linked list** is the opposite. Splicing a new node into a list, once you are holding the node it should follow, costs $\Theta(1)$. But *getting* to that node means walking the list from one end, and that is $\Theta(n)$.

One structure is fast to read and slow to write; the other is fast to write and slow to read. Neither is good enough for the obvious real problem: a collection that is searched constantly *and* changed constantly.

The **binary search tree** is the structure that refuses to choose. It keeps the halving behavior of binary search — each comparison throws away a whole side of the remaining collection — while keeping the cheap local surgery of a linked list, because inserting a node means changing two or three pointers and nothing else. Every operation in this chapter will cost $O(h)$, where h is the height of the tree, and the entire second half of the chapter is about one question: **how do we stop h from growing?**

6.2 The Vocabulary of Trees

Before anything else we need words. The reader who has never met a tree in computing before should read this section slowly; everything afterward depends on it, and none of it is difficult.

A **tree** is a collection of **nodes** joined by **edges**, drawn upside down: the single node at the top is the **root**, and the tree grows downward. Each node holds a **key** — the value we sort and search by — and usually some **satellite data** attached to that key, the way a laundry ticket number has a customer, a weight and a price attached to it. Throughout this chapter we draw only the key, because the key is all the structure cares about.

If an edge runs from node u down to node v, then u is the **parent** of v and v is a **child** of u. The root is the one node with no parent. A node with no children is a **leaf**; a node with at least one child is an **internal node**. Two nodes with the same parent are **siblings**.

In a **binary tree**, every node has at most two children, and the two are told apart by name: the **left child** and the **right child**. This distinction matters enormously. A node with only a right child is a different tree from a node with only a left child, even though both have one node hanging below one node.

A **path** from u to v is the sequence of nodes you pass through walking from u down to v along edges. If such a path exists, u is an **ancestor** of v and v is a **descendant** of u. Every node is an ancestor and a descendant of itself; when we want to exclude that case we say *proper* ancestor or *proper* descendant.

The **subtree rooted at** x is the node x together with every descendant of x. This is the single most useful idea in the chapter. A subtree is itself a perfectly good tree, and almost every algorithm we write will be a sentence of the form: *do something to the root, then do the same thing to the left subtree, then do the same thing to the right subtree.*

The **depth** of a node is the number of edges on the path from the root down to it. The root has depth 0; its children have depth 1; their children have depth 2. All the nodes of a given depth together form a **level**.

The **height** of a node is the number of edges on the longest path from that node down to a leaf. The **height of the tree** is the height of its root.

> **A convention you must fix in your mind.** We define the height of a single node with no children to be 0, and the height of the *empty* tree — no nodes at all — to be -1. Some books,

and the deletion figures later in this chapter (**Figure 6.21** and **Figure 6.22**), instead count *nodes* on the longest path, so that a single node has height 1 and the empty tree has height 0. The two conventions differ by exactly one everywhere. Nothing in this chapter depends on the choice, because every formula we care about uses a *difference* of two heights, and the $+1$ cancels. Pick one, write it in the margin, and never mix them inside a single program.

Finally, two facts that bound a binary tree's shape. Let n be the number of nodes and h the height.

The tallest possible binary tree with n nodes is a single chain: one node per level, n levels, height $n-1$. So $h \le n-1$.

The shortest possible tree with n nodes fills every level. Level 0 holds at most 1 node, level 1 at most 2, level 2 at most 4, and level d at most 2^d nodes. A tree of height h therefore holds at most

$$1 + 2 + 4 + \cdots + 2^h = \sum_{d=0}^{h} 2^d = 2^{h+1} - 1$$

nodes, using the geometric-sum identity from Chapter 3. Turning that around, $n \le 2^{h+1} - 1$, hence $2^{h+1} \ge n + 1$, hence

$$h \ge \lg(n+1) - 1.$$

Since h is a whole number, $h \ge \lceil \lg(n+1) \rceil - 1$. Put the two bounds together, and call the result 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 the dream: height about $\lg n$, so about 20 steps for a million keys. The right end is the nightmare: height n, so a million steps for a million keys. A plain binary search tree can land anywhere between them, depending on nothing more than the order in which the keys happened to arrive. An AVL tree is a binary search tree that is *forced* to stay near the left end.

6.3 What a Binary Search Tree Is

Now we add the rule that turns a binary tree into a searchable one.

DEFINITION (the binary-search-tree property). A binary tree is a **binary search tree** if, for every node x in it, both of the following hold:

- every key in the left subtree of x is **less than** the key of x;
- every key in the right subtree of x is **greater than** the key of x.

We further require, in this chapter, that no two nodes hold the same key.

Read the definition again and notice the two words that do all the work: **every** and **subtree**. The rule does not say "the left *child* is smaller." It says every key anywhere below and to the left — child, grandchild, great-grandchild, all of them — is smaller. A node three levels down on the left of the root must still be smaller than the root.

The same property is often stated recursively, in four clauses: the left branch contains only keys less than the key of the node, the right branch contains only keys greater than it, both branches

must themselves be binary search trees, and no two nodes hold the same key. **Figure 6.1** is an illustration.

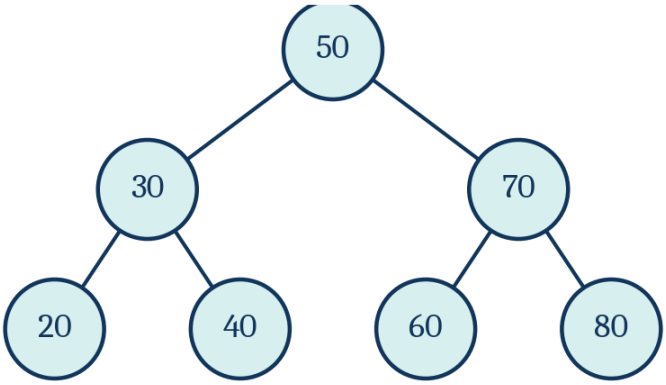


Figure 6.1 A binary search tree. The root is 50; everything in the left subtree (30,20,40) is below 50, and everything in the right subtree (70,60,80) is above it. The same test passes at every other node.

Let us check the property at every node of **Figure 6.1**, one node per line, because this is the only way to learn to see it.

- Node 50 is the root. Its left subtree holds {30,20,40}. Is every one of those less than 50? $30 < 50$ yes, $20 < 50$ yes, $40 < 50$ yes. Its right subtree holds {70,60,80}. Is every one greater than 50? $70 > 50$ yes, $60 > 50$ yes, $80 > 50$ yes. Node 50 passes.
- Node 30 has left subtree {20} and right subtree {40}. $20 < 30$ yes; $40 > 30$ yes. Passes.
- Node 70 has left subtree {60} and right subtree {80}. $60 < 70$ yes; $80 > 70$ yes. Passes.
- Nodes 20, 40, 60 and 80 are leaves. A leaf has two empty subtrees, and every statement about the members of an empty set is true by default. Leaves always pass.

Every node passes, so this is a binary search tree.

Now change one thing and watch the property break. Suppose the node 40 were instead 55. Look at node 30: its right subtree would hold {55}, and $55 > 30$, so node 30 still passes. But look at node 50: its left subtree would hold {30,20,55}, and $55 > 50$. Node 50 fails. This is the classic trap, and it is worth naming now: **a tree can satisfy the property at every parent–child pair and still not be a binary search tree.** We will write a program in §6.23 that catches exactly this bug.

> **Why the property is useful.** Stand at any node x and compare your target key k with x's key. If k is smaller, the property guarantees that k cannot be anywhere in the right subtree, so you may throw the entire right subtree away without looking at it. If k is larger, you may throw the entire left subtree away. One comparison eliminates a whole side. This is binary search, but performed on pointers instead of array indices — and, unlike array indices, pointers can be rearranged cheaply when the collection changes.

6.4 The BST Property Is Not the Heap Property

Almost every newcomer meets binary search trees and binary heaps at about the same time, and almost every newcomer mixes them up at least once. Fifteen minutes spent on the difference now saves hours of confusion later.

A **binary heap** — which you will meet in full when we study Heapsort — is also a binary tree with a rule about keys, but the rule is a completely different shape. In a **max-heap**, the rule is:

$$\text{key(parent)} \;\geq\; \text{key(child)} \quad \text{for every node that has a parent.}$$

That is the *whole* rule. It says nothing about left versus right. A node's left child and its right child are in no particular order relative to each other; either may be larger. Compare that with the binary-search-tree rule:

$$\text{every key in left}(x) \;<\; \text{key}(x) \;<\; \text{every key in right}(x). \tag{6.2}$$

The heap property is **vertical and local**: it constrains a node against its own children only, and it says nothing about left and right. The BST property Equation (6.2) is **horizontal and global**: it constrains a node against every one of its descendants, and left and right mean opposite things.

Table 6.1 sets the two side by side. Copy it out by hand once; it is the kind of table that only sticks when written.

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 $\geq$ 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

The last two rows deserve a sentence each. A heap's shape is fixed by construction — it is always filled level by level, left to right — so its height is *automatically* $\lfloor \lg n \rfloor$ and no balancing is ever needed. A binary search tree buys the ability to search for an arbitrary key, and pays for it with a shape that depends on history. That payment is the problem AVL trees solve.

> **The trap in one sentence.** If someone shows you a tree in which every parent is larger than both its children and asks you to search it for the key 37, you have no choice but to look at every node. Heaps are for repeatedly extracting the extreme; search trees are for looking things up. Using the wrong one costs you a factor of n.

6.5 Representing a Binary Search Tree in C++

A node needs three things: its key, a pointer to its left subtree, and a pointer to its right subtree. Some algorithms also want a pointer back up to the parent; we will introduce that only where it is genuinely needed, in §6.9. **Listing 6.1** declares such a node.

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;
}
```

Listing 6.1 declares the node and then wires up the seven nodes of **Figure 6.1** one pointer at a time, so that you can see a tree being made of nothing but keys and two pointers per node. It prints:

```
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 represented by the null pointer `nullptr`. This is not a detail to skim past: it is the base case of essentially every recursion in the chapter. A subtree that is not there is a subtree of height -1 containing no keys, and every claim we make about "all the keys in this subtree" is vacuously true of it.

A whole tree is then just a pointer to its root node, and `nullptr` is a perfectly good whole tree with nothing in it.

6.6 Walking the Tree: Three Traversals

To *traverse* a tree is to visit every node exactly once. For a binary tree there are three classical orders, and they differ only in *when* the root of a subtree is visited relative to its two subtrees.

> **DEFINITION (the three depth-first traversals).** To traverse the subtree rooted at x:

- **inorder** — traverse the left subtree, then visit x, then traverse the right subtree;
- **preorder** — visit x, then traverse the left subtree, then traverse the right subtree;
- **postorder** — traverse the left subtree, then traverse the right subtree, then visit x.

> In every case, traversing the empty tree does nothing at all.

The names describe where the root falls: *pre*-order puts it before its subtrees, *in*-order puts it between them, *post*-order puts it after them. In all three the left subtree is dealt with before the right.

Let us trace all three on the tree of **Figure 6.1**, by hand, completely.

Inorder, starting at the root 50.

1. Visit the left subtree of 50, which is rooted at 30. Hold 50 in reserve.
2. At 30: visit its left subtree, rooted at 20. Hold 30 in reserve.
3. At 20: its left subtree is empty, so nothing happens. Now output **20**. Its right subtree is empty, so nothing happens. Return.
4. Back at 30: its left subtree is done. Output **30**. Now its right subtree, rooted at 40.
5. At 40: left empty; output **40**; right empty. Return.
6. Back at 50: its left subtree is finished. Output **50**. Now its right subtree, rooted at 70.
7. At 70: left subtree first, rooted at 60. Output **60**. Then output **70**. Then the right subtree, rooted at 80: output **80**.

The output is 20,30,40,50,60,70,80 — sorted.

Preorder, on the same tree. The root is emitted first at every level: 50; then the whole left subtree in preorder, which emits 30, then 20, then 40; then the whole right subtree in preorder, which emits 70, then 60, then 80. The output is 50,30,20,40,70,60,80.

Postorder. Every root waits for both of its subtrees. At 50: the left subtree first. Inside it, at 30: left subtree gives 20, right subtree gives 40, then 30 itself. So the left half emits 20,40,30. Then the right half emits 60,80,70. Then finally 50. The output is 20,40,30,60,80,70,50.

Listing 6.2 builds the tree and prints all three walks, and also draws the tree sideways so that you can check the shape against the figure. Turn the printed picture ninety degrees clockwise and it is **Figure 6.1**.

Listing 6.2 The three traversals in C++. Save as `traversal.cpp`, compile with `g++ -std=c++17 -O2 -o traversal traversal.cpp`, and run `./traversal`.

```
// Program: the three depth-first traversals of a binary search tree.
// Builds the tree of the first figure and walks it three ways.
#include <iostream>
#include <string>
#include <vector>

struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

// Insert one key, keeping the binary-search-tree property. Duplicates ignored.
Node* insertKey(Node* root, int key) {
    if (root == nullptr) return new Node(key);
    if (key < root->key)      root->left  = insertKey(root->left,  key);
    else if (key > root->key) root->right = insertKey(root->right, key);
    return root;
}

void inorder(Node* x, std::vector<int>& out) {
    if (x == nullptr) return;
    inorder(x->left, out);
    out.push_back(x->key);
    inorder(x->right, out);
}

void preorder(Node* x, std::vector<int>& out) {
    if (x == nullptr) return;
    out.push_back(x->key);
    preorder(x->left, out);
    preorder(x->right, out);
}

void postorder(Node* x, std::vector<int>& out) {
    if (x == nullptr) return;
    postorder(x->left, out);
    postorder(x->right, out);
    out.push_back(x->key);
}
```

Listing 6.2 (Continued) *The three traversals in C++. Save as traversal.cpp, compile with g++ -std=c++17 -O2 -o traversal traversal.cpp, and run ./traversal.*

```
void printSeq(const std::string& label, const std::vector<int>& v) {
    std::cout << label;
    for (size_t i = 0; i < v.size(); ++i) {
        if (i) std::cout << ", ";
        std::cout << v[i];
    }
    std::cout << "\n";
}

// A sideways picture of the tree: right subtree on top, root, then left subtree.
void draw(Node* x, int depth) {
    if (x == nullptr) return;
    draw(x->right, depth + 1);
    std::cout << std::string(4 * depth, ' ') << x->key << "\n";
    draw(x->left, depth + 1);
}

int main() {
    Node* root = nullptr;
    int keys[] = {50, 30, 70, 20, 40, 60, 80};
    for (int k : keys) root = insertKey(root, k);

    std::cout << "The tree, drawn sideways (root at the left margin):\n";
    draw(root, 0);
    std::cout << "\n";

    std::vector<int> a, b, c;
    inorder(root, a);
    preorder(root, b);
    postorder(root, c);
    printSeq("inorder   : ", a);
    printSeq("preorder  : ", b);
    printSeq("postorder : ", c);

    bool sorted = true;
    for (size_t i = 1; i < a.size(); ++i)
        if (a[i - 1] >= a[i]) sorted = false;
    std::cout << "\ninorder output is sorted: " << (sorted ? "yes" : "no") << "\n";
    return 0;
}
```

Running it prints exactly this:

```
The tree, drawn sideways (root at the left margin):
        80
    70
        60
50
        40
    30
        20

inorder   : 20, 30, 40, 50, 60, 70, 80
preorder  : 50, 30, 20, 40, 70, 60, 80
postorder : 20, 40, 30, 60, 80, 70, 50

inorder output is sorted: yes
```

Every one of the three lines matches the hand trace above.

6.6.1 Why the inorder walk is sorted — a proof

The program says "yes", but a program saying yes is evidence, not proof. Here is the proof, and it is short because the definition does all the work.

Claim. If T is a binary search tree, an inorder traversal of T outputs its keys in strictly increasing order.

Proof, by induction on the number of nodes n in T.

Base case, $n = 0$. The traversal outputs nothing. The empty sequence is trivially increasing.

Inductive step. Assume the claim holds for every binary search tree with fewer than n nodes, and let T have $n \geq 1$ nodes with root x. Write L for the left subtree of x and R for the right subtree.

First, L and R are themselves binary search trees. This is immediate: the BST property is a statement about *every* node, so it holds in particular at every node of L and every node of R, and a node's subtrees inside L are the same subtrees they were inside T. Both L and R have fewer than n nodes, since x belongs to neither. So the induction hypothesis applies to both.

The traversal outputs, in order: the inorder walk of L, then the key of x, then the inorder walk of R. By the induction hypothesis the first block is increasing and the third block is increasing. It remains to check the two joints.

At the first joint: the last key printed from L is some key of L, and by the BST property at x, *every* key of L is less than key(x). So the last key of the first block is less than key(x).

At the second joint: the first key printed from R is some key of R, and by the BST property at x, every key of R is greater than key(x). So key(x) is less than the first key of the third block.

The whole output is therefore increasing, which completes the induction. ■

> **A free sorting algorithm.** Insert n keys into a binary search tree, then walk it inorder. The keys come out sorted. This is *tree sort*. Its cost is the cost of n insertions plus one $\Theta(n)$ walk, so it is $O(nh)$ — which is $\Theta(n\lg n)$ when the tree stays short, and $\Theta(n^2)$ when it does not. You now have a concrete reason to care about the height of a tree: it is the difference between an excellent sorting algorithm and a terrible one.

Table 6.2 records what each walk is actually used for, because all three do show up in practice.

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

Each walk visits every node once and does $O(1)$ work there, so each costs $\Theta(n)$ — regardless of shape. Traversal is the one operation in this chapter whose cost does *not* depend on the height.

6.7 Searching

Searching is the operation the structure was named for, and it is four lines long.

> **TREE-SEARCH(x, k).** To look for the key k in the subtree rooted at x:

1. If x is empty, the key is not present — report failure.
2. If k equals the key of x, report success and return x.
3. If k is less than the key of x, search the left subtree of x.
4. Otherwise search the right subtree of x.

Why is step 3 allowed to ignore the right subtree completely? Because the BST property says *every* key in the right subtree of x is greater than $\text{key}(x)$, and $k < \text{key}(x)$, so no key in the right subtree can equal k. Step 4 is the mirror image. Each comparison eliminates one entire side.

Let us trace TREE-SEARCH on the tree of **Figure 6.2**: root 20, with left child 8 and right child 22; node 8 has children 4 and 12; node 12 has children 10 and 14.

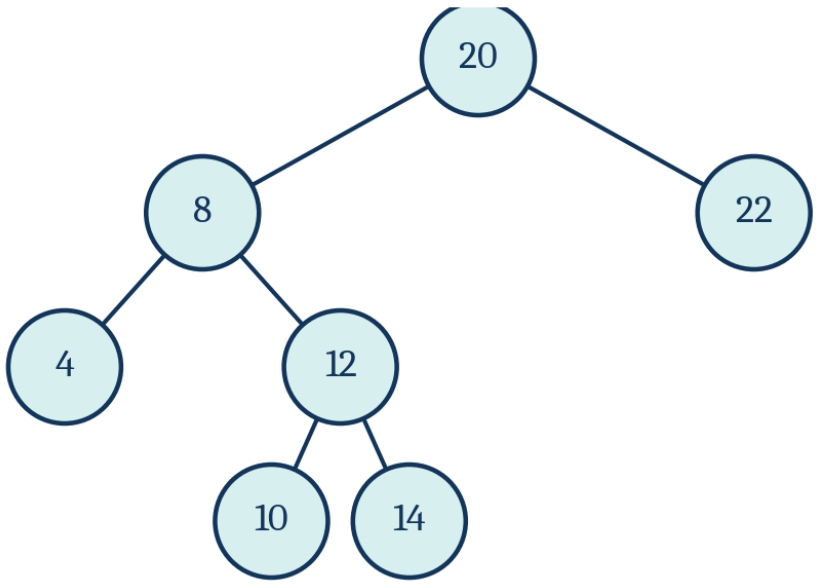


Figure 6.2 A tree for practicing inorder successors. Walking it inorder gives 4,8,10,12,14,20,22, and reading that list is the easiest way to check any successor or predecessor claim.

Trace of SEARCH(10)**.**

1. Start at the root, 20. Is $10 = 20$? No. Is $10 < 20$? Yes, so go left.
2. Now at 8. Is $10 = 8$? No. Is $10 < 8$? No, so go right.
3. Now at 12. Is $10 = 12$? No. Is $10 < 12$? Yes, so go left.
4. Now at 10. Is $10 = 10$? Yes. Found, after visiting four nodes: $20 \to 8 \to 12 \to 10$.

Trace of SEARCH(13), a key that is absent.

1. At 20: $13 < 20$, go left.
2. At 8: $13 > 8$, go right.
3. At 12: $13 > 12$, go right.
4. At 14: $13 < 14$, go left.
5. The left child of 14 is empty. Report failure, after visiting four nodes.

Notice what the failed search told us anyway: 13 would belong exactly where we fell off the tree, as the left child of 14. That is not a coincidence, and §6.10 exploits it.

Cost. Each step moves strictly one level down the tree and does a constant amount of comparing, so the number of steps is at most the length of the longest root-to-leaf path plus one, which is $h + 1$. The running time is $O(h)$.

6.8 Minimum and Maximum

Where is the smallest key? Ask the property. Every key smaller than the root lives in the left subtree, so if the left subtree is not empty the minimum is in there. Repeat. The recursion only stops when there is no left subtree left to go into.

> **TREE-MINIMUM**(x)**.** While x has a left child, replace x by its left child. Return x.
>
> **TREE-MAXIMUM**(x)**.** While x has a right child, replace x by its right child. Return x.

On **Figure 6.2**: MINIMUM starts at 20, goes left to 8, goes left to 4; node 4 has no left child, so 4 is the minimum. Three nodes visited. MAXIMUM starts at 20, goes right to 22; node 22 has no right child, so 22 is the maximum. Two nodes visited.

Both walk a single downward path, so both cost $O(h)$.

> **Do not confuse "leftmost" with "deepest on the left."** The minimum is the node you reach by always turning left. It need not be the deepest node in the tree, and it need not be a leaf — it may well have a right child. In **Figure 6.2**, if node 4 had a right child 6, then 4 would still be the minimum and would still not be a leaf.

6.9 Successor and Predecessor

> **DEFINITION.** The **inorder successor** of a node x in a binary search tree is the node holding the smallest key that is still larger than key(x). Equivalently, it is the node that comes immediately after x in the inorder walk. The largest key in the tree has no successor. The **inorder predecessor** is the mirror image: the node with the largest key smaller than key(x), the node immediately before x in the inorder walk, and the smallest key has none.

Three examples, which we can verify by simply reading the inorder walk of **Figure 6.2**, namely 4,8,10,12,14,20,22:

- the successor of 8 is 10 — and indeed 10 follows 8 in the list;
- the successor of 10 is 12;
- the successor of 14 is 20.

Finding the successor without writing out the whole walk splits into two cases, and the split is the interesting part.

Case 1: x has a right subtree. Then the successor is the *minimum of that right subtree*. Why? Every key in the right subtree is greater than key(x), so all of them are candidates; the successor is the smallest candidate; the smallest key in a subtree is found by TREE-MINIMUM. And no key outside the right subtree can sneak in between: any key greater than key(x) that is not in the right subtree lies above x, and every such key is greater than *everything* in x's right subtree.

On **Figure 6.2**, take $x = 8$. Its right subtree is rooted at 12. TREE-MINIMUM of that subtree goes $12 \to 10$ and stops. The successor of 8 is 10. ✓

Case 2: x has no right subtree. Then the successor is the lowest ancestor of x whose *left* subtree contains x. In practice: climb upward from x; every time you arrive at a parent from its *right* child, keep climbing; the first time you arrive at a parent from its *left* child, that parent is the successor. If you climb off the top of the tree, x was the maximum and has no successor.

Why does this work? If you came up from a left child, the parent's key is greater than everything in the subtree you just left, including key(x), and it is the first such key you met going up — so it is

the smallest key greater than key(x). If you came up from a right child, the parent's key is *smaller* than key(x), so it is not a candidate at all.

On **Figure 6.2**, take $x = 14$. It has no right subtree. Climb: 14 is the right child of 12 — came up from the right, keep going. 12 is the right child of 8 — came up from the right, keep going. 8 is the *left* child of 20 — stop. The successor of 14 is 20. ✓

Take $x = 22$. No right subtree. Climb: 22 is the right child of 20 — keep going. Above 20 there is nothing. So 22 has no successor. ✓ It is the maximum.

Case 2 needs a way to move *upward*, which our three-field node cannot do. There are two standard fixes. The first is to add a `parent` pointer to every node, which is what **Listing 6.3** does. The second is to carry a "best candidate so far" down from the root during an ordinary search — every time you turn left, the node you are leaving becomes the new best candidate. The second needs no extra memory and is what we will use in the Profitina Laundry case study in §6.22.

Listing 6.3 implements all five query operations and prints the path each one walks.

Listing 6.3 Search, minimum, maximum, successor and predecessor, each reporting the path it walked. Compile with `g++ -std=c++17 -O2 -o query query.cpp`.

```
// Program: the five query operations of a binary search tree, each one
// printing the path it walked, on the successor tree of Section 6.7.
#include <iostream>
#include <string>
#include <vector>

struct Node {
    int key;
    Node* left;
    Node* right;
    Node* parent;                       // needed by SUCCESSOR and PREDECESSOR
    Node(int k) : key(k), left(nullptr), right(nullptr), parent(nullptr) {}
};

Node* insertKey(Node* root, int key) {
    Node* z = new Node(key);
    Node* y = nullptr;                  // y trails x, so it ends at z's parent
    Node* x = root;
    while (x != nullptr) {
        y = x;
        if (key < x->key) x = x->left;
        else              x = x->right;
    }
    z->parent = y;
    if (y == nullptr)            return z;      // the tree was empty
    if (key < y->key) y->left  = z;
    else              y->right = z;
    return root;
}

Node* treeSearch(Node* x, int k, std::vector<int>& path) {
    while (x != nullptr && x->key != k) {
        path.push_back(x->key);
        if (k < x->key) x = x->left;
        else            x = x->right;
    }
    if (x != nullptr) path.push_back(x->key);
    return x;
}

Node* treeMinimum(Node* x, std::vector<int>& path) {
    while (x->left != nullptr) { path.push_back(x->key); x = x->left; }
    path.push_back(x->key);
    return x;
}
```

Listing 6.3 (Continued) *Search, minimum, maximum, successor and predecessor, each reporting the path it walked. Compile with g++ -std=c++17 -O2 -o query query.cpp.*

```
Node* treeMaximum(Node* x, std::vector<int>& path) {
    while (x->right != nullptr) { path.push_back(x->key); x = x->right; }
    path.push_back(x->key);
    return x;
}

Node* treeSuccessor(Node* x) {
    if (x->right != nullptr) {          // case 1: smallest key in the right subtree
        Node* m = x->right;
        while (m->left != nullptr) m = m->left;
        return m;
    }
    Node* y = x->parent;                // case 2: climb until we come up from a left child
    while (y != nullptr && x == y->right) { x = y; y = y->parent; }
    return y;
}

Node* treePredecessor(Node* x) {
    if (x->left != nullptr) {
        Node* m = x->left;
        while (m->right != nullptr) m = m->right;
        return m;
    }
    Node* y = x->parent;
    while (y != nullptr && x == y->left) { x = y; y = y->parent; }
    return y;
}

void show(const std::string& label, const std::vector<int>& path) {
    std::cout << label;
    for (size_t i = 0; i < path.size(); ++i) {
        if (i) std::cout << " -> ";
        std::cout << path[i];
    }
    std::cout << "   (" << path.size() << " nodes visited)\n";
}

int main() {
    Node* root = nullptr;
    int keys[] = {20, 8, 22, 4, 12, 10, 14};
    for (int k : keys) root = insertKey(root, k);
```

Listing 6.3 (Continued) *Search, minimum, maximum, successor and predecessor, each reporting the path it walked. Compile with g++ -std=c++17 -O2 -o query query.cpp.*

```
    std::vector<int> p;
    treeSearch(root, 10, p);  show("SEARCH(10)  path: ", p);
    p.clear();
    treeSearch(root, 13, p);  show("SEARCH(13)  path: ", p);
    p.clear();
    treeMinimum(root, p);     show("MINIMUM     path: ", p);
    p.clear();
    treeMaximum(root, p);     show("MAXIMUM     path: ", p);

    std::cout << "\n";
    int probe[] = {8, 10, 14, 4, 22};
    for (int k : probe) {
        std::vector<int> q;
        Node* x = treeSearch(root, k, q);
        Node* s = treeSuccessor(x);
        Node* t = treePredecessor(x);
        std::cout << "key " << k
                  << ": successor = "   << (s ? std::to_string(s->key) :
std::string("none"))
                  << ", predecessor = " << (t ? std::to_string(t->key) :
std::string("none"))
                  << "\n";
    }
    return 0;
}
```

The program prints:

```
SEARCH(10)  path: 20 -> 8 -> 12 -> 10   (4 nodes visited)
SEARCH(13)  path: 20 -> 8 -> 12 -> 14   (4 nodes visited)
MINIMUM     path: 20 -> 8 -> 4   (3 nodes visited)
MAXIMUM     path: 20 -> 22   (2 nodes visited)

key 8: successor = 10, predecessor = 4
key 10: successor = 12, predecessor = 8
key 14: successor = 20, predecessor = 12
key 4: successor = 8, predecessor = none
key 22: successor = none, predecessor = 20
```

Every path matches the hand trace, and the three successor claims match the examples of §6.9 exactly. Note the last two lines: the minimum 4 has no predecessor and the maximum 22 has no successor, as the definition requires.

Cost. Case 1 walks down one path; case 2 walks up one path. Neither ever changes direction. So SUCCESSOR and PREDECESSOR both cost $O(h)$.

6.10 Insertion

A failed search tells you where a key *would* be. Insertion is nothing more than doing that failed search and then attaching a new node where you fell off.

> **TREE-INSERT**(T, k)**.** Walk down from the root exactly as TREE-SEARCH would, remembering the node you are standing on as you leave it. When you step off the bottom of the tree — that is, when the child you want to visit is empty — create a new node holding k and make it that child. A newly inserted node is always a leaf.

Three facts sum it up: a new element is always inserted as a leaf; the place is found by descending from the root until an empty position is reached; and the new node is then attached as a child there. **Figure 6.3** illustrates `Insert(50)`.

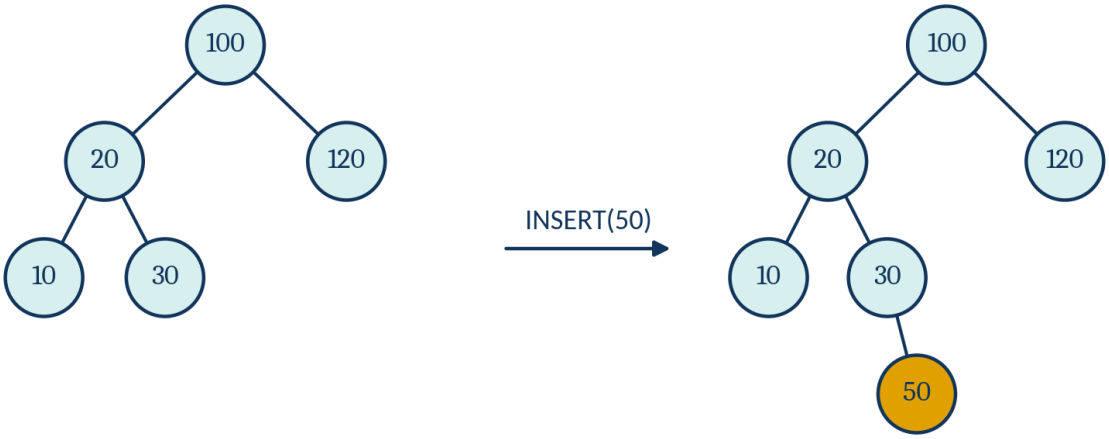


Figure 6.3 An insertion example. The key 50 is threaded down the tree — left at 100, right at 20, right at 30 — and lands as the new right child of 30, shown in amber. Nothing else in the tree moves.

Trace `Insert(50)` on the left-hand tree of **Figure 6.3**, one comparison per line.

1. Start at the root, 100. Is $50 < 100$? Yes. Go left. Remember 100.
2. Now at 20. Is $50 < 20$? No. Go right. Remember 20.
3. Now at 30. Is $50 < 30$? No. Go right. Remember 30.
4. The right child of 30 is empty. Stop. Create a node holding 50 and make it the right child of 30.

The result is the right-hand tree of **Figure 6.3**, with 50 shaded to mark it as new.

Check the property afterward, because this is the step people skip. Node 30 now has right subtree $\{50\}$ and $50 > 30$ ✓. Node 20 now has right subtree $\{30,50\}$ and both are greater than 20 ✓. Node 100 now has left subtree $\{20,10,30,50\}$ and all four are less than 100 ✓. The property survives, and the reason it survives is precisely that the descent path was chosen by the same comparisons the property makes.

> **Why the new node is always a leaf.** The descent only stops when it wants to step into an empty subtree. So the position we fill was empty, and a node placed in an empty position has no children. This single fact is what makes insertion cheap: we never have to move an existing node, only attach to one.

Cost. The descent is a root-to-leaf path, so insertion costs $O(h)$, plus $O(1)$ to allocate and attach the node.

Order matters, enormously. Insert the keys 1,2,3,4,5 into an empty tree in that order and every one of them goes right, producing a chain of height 4. Insert the same five keys in the order 3,1,4,2,5 and you get a tree of height 2. Same keys, same algorithm, completely different structure. Hold that thought until §6.13.

6.11 Deletion: the Three Cases

Deletion is the only operation in this chapter with real case analysis, and it is organized the standard way: by how many children the doomed node has. There are three possibilities — none, one, or two — and they are genuinely different problems.

6.11.1 Case 1 — the node is a leaf

If the node has no children, nothing depends on it. Detach it from its parent and free it.

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.

In **Figure 6.4** the tree is 50 at the root, with 30 and 70 below it, and leaves 20,40,60,80. To delete 20:

1. Search for 20: at 50, go left; at 30, go left; found 20.
2. Node 20 has no left child and no right child, so it is a leaf.
3. Set the left child of its parent 30 to empty, and free the node.

Afterward the inorder walk is 30,40,50,60,70,80 — the sorted list with 20 removed, and nothing else disturbed.

6.11.2 Case 2 — the node has exactly one child

If the node has exactly one child, then the whole subtree hanging below it is on one side. Promote that subtree: make it the child of the deleted node's parent, in the deleted node's old position.

It can also be phrased as "copy the child into the node to be deleted, then delete the child", which is the same surgery seen from the other end and is equally correct. We will phrase it as promotion, because promotion is what the C++ code does.

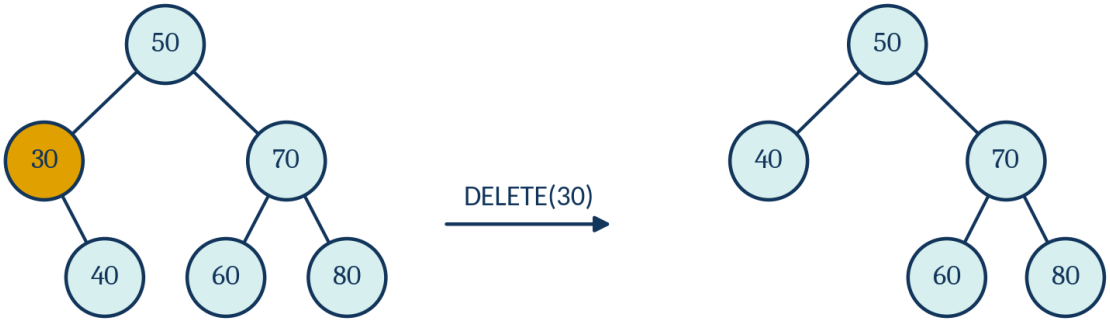


Figure 6.5 Case 2 of deletion: node 30 has only a right child, 40. Deleting 30 promotes 40 into its place.

Continuing from the tree of **Figure 6.5** — which is the previous tree after 20 was removed — delete 30:

1. Search for 30: at 50, go left; found 30.
2. Node 30 has no left child, but it does have a right child 40. Exactly one child.
3. Make 40 the left child of 50, in place of 30. Free node 30.

Why is the property preserved? Because every key in the promoted subtree was already on the correct side of every ancestor. The subtree rooted at 40 sat below 30, which sat to the left of 50; so every key in it was already less than 50. Moving it one level up does not change which ancestors it is below, only how far below it sits.

Afterward the inorder walk is 40,50,60,70,80.

6.11.3 Case 3 — the node has two children

This is the hard case, and the trick is to refuse to solve it. If a node has two children we do not remove that node at all. We overwrite its key with a key that may legally sit there, and then delete *that* key's node, which is guaranteed to be an easier case.

Which key may legally sit there? The **inorder successor** — the smallest key in the right subtree. The recipe, stated exactly: find the inorder successor, which is min(right subtree), copy it into the node to be deleted, then delete the inorder successor.

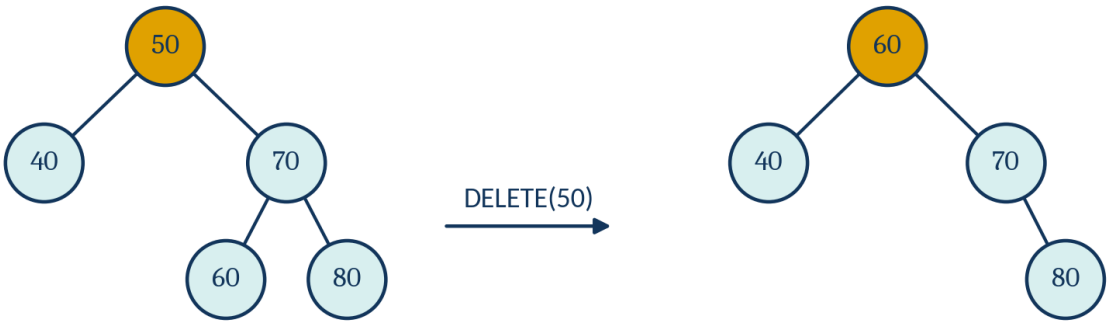


Figure 6.6 Case 3 of deletion: node 50 has two children. Its inorder successor is 60, the smallest key in its right subtree. The value 60 is copied up into the root and the original node 60 is then deleted, which is a case-1 deletion.

Continuing from the tree of **Figure 6.6** — the previous tree after 30 was removed, so the root 50 has left child 40 and right child 70, and 70 has children 60 and 80 — delete 50:

1. Search for 50: it is the root. Found immediately.
2. Node 50 has a left child (40) and a right child (70). Two children.
3. Find the inorder successor of 50: take one step right to 70, then walk left as far as possible. Node 70's left child is 60; node 60 has no left child. The successor is 60.
4. Copy the key 60 into the root node, overwriting 50. The tree momentarily holds two nodes labeled 60.
5. Delete the *original* node 60 from the right subtree. It is a leaf, so this is case 1: detach it from its parent 70.

Afterward the inorder walk is 40,60,70,80, exactly as the right-hand tree of **Figure 6.6** shows.

Why the successor is a legal replacement. Call the doomed node x and its successor s. Every key in x's left subtree is less than key(x), and key(x) < key(s), so every key in the left subtree is less than key(s): the left side is happy. Every key in x's right subtree is greater than key(x), and key(s) is the *smallest* of them, so every remaining key in the right subtree is greater than key(s): the right side is happy. Putting key(s) at x's position therefore satisfies the BST property at x, and no other node's situation changed.

Why the recursive deletion is easier. The successor s is the leftmost node of x's right subtree, so by construction s has no left child. A node with no left child has zero or one child — case 1 or case 2. The recursion therefore stops immediately; it can never hit case 3 again.

The predecessor works just as well. The inorder *predecessor* — max(left subtree) — may be used instead, by the mirror argument. It has no right child, so it is also an easy case. Some implementations alternate between successor and predecessor to avoid the left-leaning drift that always choosing the successor tends to cause. In **Figure 6.6**, choosing the predecessor would

have promoted 40 instead of 60, giving the equally valid tree 40 at the root with right child 70 and grandchildren 60,80.

Table 6.3 collects the 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

Cost. Finding the node costs $O(h)$. Cases 1 and 2 then do constant work. Case 3 performs one further descent to find the successor, which is another $O(h)$. So deletion costs $O(h)$ overall.

6.12 A Complete Binary Search Tree in C++

Listing 6.4 is a full, runnable binary search tree: insert, search, minimum, maximum, height, inorder listing, and deletion in all three cases. Its `main` replays the three deletion examples of §6.11 in order, so the printed output can be checked line by line against **Figure 6.4**, **Figure 6.5** and **Figure 6.6**.

Listing 6.4 A complete binary search tree. Compile with `g++ -std=c++17 -O2 -o bst bst.cpp` and run `./bst`.

```
// A complete binary search tree in C++: insert, search, minimum, maximum,
// successor, predecessor, delete (all three cases), and an inorder listing.
// The main() function replays the three deletion examples of Section 6.11 in order.
#include <iostream>
#include <string>
#include <vector>

class BST {
public:
    BST() : root_(nullptr) {}
    ~BST() { destroy(root_); }

    void insert(int key)           { root_ = insertRec(root_, key); }
    bool contains(int key) const   { return find(root_, key) != nullptr; }
    void remove(int key)           { root_ = removeRec(root_, key); }

    int minimum() const { const Node* x = root_; while (x->left)  x = x->left;  return x-
>key; }
    int maximum() const { const Node* x = root_; while (x->right) x = x->right; return x-
>key; }

    int height() const { return heightRec(root_); }   // empty tree has height -1

    std::vector<int> inorder() const {
        std::vector<int> out;
        inorderRec(root_, out);
        return out;
    }

    void print(const std::string& label) const {
        std::cout << label;
        std::vector<int> v = inorder();
        for (size_t i = 0; i < v.size(); ++i) {
            if (i) std::cout << ", ";
            std::cout << v[i];
        }
        std::cout << "   [height " << height() << "]\n";
    }

private:
    struct Node {
        int key;
        Node* left;
        Node* right;
        Node(int k) : key(k), left(nullptr), right(nullptr) {}
    };
    Node* root_;
```

Listing 6.4 (Continued) *A complete binary search tree. Compile with g++ -std=c++17 -O2 -o bst bst.cpp and run ./bst.*

```
    static void destroy(Node* x) {
        if (!x) return;
        destroy(x->left);
        destroy(x->right);
        delete x;
    }

    static Node* insertRec(Node* x, int key) {
        if (!x) return new Node(key);
        if      (key < x->key) x->left  = insertRec(x->left,  key);
        else if (key > x->key) x->right = insertRec(x->right, key);
        return x;                                   // equal key: do nothing
    }

    static const Node* find(const Node* x, int key) {
        while (x && x->key != key) x = (key < x->key) ? x->left : x->right;
        return x;
    }

    static int heightRec(const Node* x) {
        if (!x) return -1;
        int l = heightRec(x->left), r = heightRec(x->right);
        return 1 + (l > r ? l : r);
    }

    static void inorderRec(const Node* x, std::vector<int>& out) {
        if (!x) return;
        inorderRec(x->left, out);
        out.push_back(x->key);
        inorderRec(x->right, out);
    }

    static Node* minNode(Node* x) { while (x->left) x = x->left; return x; }

    static Node* removeRec(Node* x, int key) {
        if (!x) return nullptr;                     // key absent: nothing to do
        if      (key < x->key) { x->left  = removeRec(x->left,  key); return x; }
        else if (key > x->key) { x->right = removeRec(x->right, key); return x; }
```

Listing 6.4 (Continued) *A complete binary search tree. Compile with g++ -std=c++17 -O2 -o bst bst.cpp and run ./bst.*

```
        // x is the node to delete.
        if (!x->left && !x->right) {                // case 1: no child
            delete x;
            return nullptr;
        }
        if (!x->left) {                             // case 2a: only a right child
            Node* c = x->right;
            delete x;
            return c;
        }
        if (!x->right) {                            // case 2b: only a left child
            Node* c = x->left;
            delete x;
            return c;
        }
        Node* s = minNode(x->right);                // case 3: inorder successor
        x->key = s->key;                            // copy its key up
        x->right = removeRec(x->right, s->key);     // then delete it from the right subtree
        return x;
    }
};

int main() {
    BST t;
    for (int k : {50, 30, 70, 20, 40, 60, 80}) t.insert(k);
    t.print("start                 : ");

    std::cout << "contains(40) = " << (t.contains(40) ? "true" : "false")
              << ", contains(45) = " << (t.contains(45) ? "true" : "false") << "\n";
    std::cout << "minimum = " << t.minimum() << ", maximum = " << t.maximum() << "\n\n";

    t.remove(20);  t.print("case 1, delete 20     : ");
    t.remove(30);  t.print("case 2, delete 30     : ");
    t.remove(50);  t.print("case 3, delete 50     : ");
    return 0;
}
```

The program prints:

```
start                 : 20, 30, 40, 50, 60, 70, 80   [height 2]
contains(40) = true, contains(45) = false
minimum = 20, maximum = 80

case 1, delete 20     : 30, 40, 50, 60, 70, 80   [height 2]
case 2, delete 30     : 40, 50, 60, 70, 80   [height 2]
case 3, delete 50     : 40, 60, 70, 80   [height 2]
```

Three remarks on the code, because each one is a place beginners lose a day.

The functions return the new root of the subtree. `insertRec` and `removeRec` both take a subtree and hand back a subtree, and the caller writes the result into the correct child pointer: `x->left = insertRec(x->left, key)`. This idiom removes the need to keep a parent pointer, and it removes an entire family of bugs in which a node is changed but its parent still points at the old one. Every recursive tree function in this chapter uses it.

Case 2 is written twice. `if (!x->left)` handles a node with only a right child and `if (!x->right)` handles a node with only a left child. They could be collapsed into one branch, and in the AVL code later we do exactly that. Written out separately here, they are easier to match against **Figure 6.5**.

Case 3 copies a key, not a node. After `x->key = s->key`, the node object at position x is still the same object in memory; only the integer inside it changed. If the node also carried satellite data — a customer name, a weight — that data would have to be copied too, or the tree would hold a successor's key against a deleted customer's record. In production code it is usually safer to relink the successor node itself rather than copy its contents.

6.13 When the Tree Goes Bad: h Can Reach n

Every cost in this chapter so far has been $O(h)$. That is a genuine bound but it is not a *promise*, because nothing so far stops h from being large.

Consider inserting the keys 10,20,30,40,50 into an empty tree in that order.

1. 10 becomes the root.
2. $20 > 10$, so it goes right. The tree is $10 \to 20$.
3. $30 > 10$, go right; $30 > 20$, go right. The tree is $10 \to 20 \to 30$.
4. 40 likewise ends up below 30.
5. 50 likewise ends up below 40.

The result is a chain: five nodes, five levels, height 4. Every node except the last has exactly one child, and that child is always the right one. This shape is called a **degenerate** or **skewed** tree, and it is exactly a linked list wearing a tree's clothes. Searching it costs $\Theta(n)$. This is precisely the failure to fear: most binary-search-tree operations take time proportional to the tree's height, and that height becomes $n - 1$ for a skewed tree.

This is not a rare accident. Sorted input is the most common input there is. Records arrive in date order, invoice numbers are issued in increasing order, log lines are appended in timestamp order, a database exports its rows in primary-key order. Every one of those, fed straight into a plain binary search tree, produces a chain.

Listing 6.5 measures the damage. It builds two trees of the same n keys — once in ascending order, once shuffled — and reports the height of each and the average number of nodes an insertion had to look at.

Listing 6.5 Measuring what insertion order does to a binary search tree. Compile with `g++ -std=c++17 -O2 -o degenerate degenerate.cpp`.

```
// What the order of insertion does to the height of a binary search tree.
// Sorted input makes a chain of n nodes; shuffled input makes a short tree.
#include <iostream>
#include <iomanip>
#include <random>
#include <vector>
#include <algorithm>
#include <cmath>

struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

// Iterative insert, and it returns how many nodes it had to look at.
int insertCounting(Node*& root, int key) {
    int probes = 0;
    if (!root) { root = new Node(key); return 1; }
    Node* x = root;
    for (;;) {
        ++probes;
        if (key < x->key) {
            if (!x->left) { x->left = new Node(key); return probes; }
            x = x->left;
        } else {
            if (!x->right) { x->right = new Node(key); return probes; }
            x = x->right;
        }
    }
}

int height(Node* x) {
    if (!x) return -1;
    int l = height(x->left), r = height(x->right);
    return 1 + (l > r ? l : r);
}

void destroy(Node* x) { if (!x) return; destroy(x->left); destroy(x->right); delete x; }
```

Listing 6.5 (Continued) *Measuring what insertion order does to a binary search tree. Compile with g++ -std=c++17 -O2 -o degenerate degenerate.cpp.*

```
// Build a tree from the given order; report its height and the search cost.
void trial(const char* name, std::vector<int> order, int n) {
    Node* root = nullptr;
    long long totalProbes = 0;
    for (int k : order) totalProbes += insertCounting(root, k);
    int h = height(root);
    std::cout << std::left << std::setw(20) << name
              << " n = " << std::setw(7) << n
              << " height = " << std::setw(8) << h
              << " average probes per insert = "
              << std::fixed << std::setprecision(2)
              << (double)totalProbes / n << "\n";
    destroy(root);
}

int main() {
    std::mt19937 rng(12345);                  // fixed seed, so the run repeats
    for (int n : {1000, 4000, 16000}) {
        std::vector<int> sorted(n);
        for (int i = 0; i < n; ++i) sorted[i] = i;
        std::vector<int> shuffled = sorted;
        std::shuffle(shuffled.begin(), shuffled.end(), rng);

        trial("sorted input",   sorted,   n);
        trial("shuffled input", shuffled, n);
        std::cout << "   lg(n) = " << std::fixed << std::setprecision(2)
                  << std::log2((double)n) << "\n\n";
    }
    return 0;
}
```

The program prints:

```
sorted input         n = 1000    height = 999      average probes per insert = 499.50
shuffled input       n = 1000    height = 21       average probes per insert = 10.55
   lg(n) = 9.97

sorted input         n = 4000    height = 3999     average probes per insert = 1999.50
shuffled input       n = 4000    height = 25       average probes per insert = 13.48
   lg(n) = 11.97

sorted input         n = 16000   height = 15999    average probes per insert = 7999.50
shuffled input       n = 16000   height = 32       average probes per insert = 16.03
   lg(n) = 13.97
```

Read those numbers slowly, because they are the justification for the entire second half of this chapter.

With sorted input the height is always exactly $n-1$: a perfect chain, every time. The average number of probes per insertion is $(n-1)/2$, which for $n = 16000$ is 7999.5. With shuffled input the height at $n = 16000$ is 32, and the average probe count is 16.03 — while $\lg 16000 \approx 13.97$. Shuffled input therefore costs about 16 probes where sorted input costs about 8000: a factor of roughly **five hundred**, on the same data, with the same code, differing only in arrival order.

> **The random case is fine; the random case is not the problem.** It can be proved that a binary search tree built by inserting n distinct keys in uniformly random order has expected height $\Theta(\lg n)$. The constant is larger than one: for very large n the expected height approaches $4.311 \ln n \approx 2.99 \lg n$, and a correction term keeps it somewhat lower at moderate sizes. That is why the measured heights above are 21, 25 and 32 rather than the 10, 12 and 14 of a perfectly balanced tree. The trouble is that real data is almost never in random order, and you rarely

control the order. A structure whose performance depends on your luck is a structure that will eventually fail on the day it matters. We want a *guarantee*, not an expectation.

6.14 AVL Trees: Balance as an Invariant

In 1962 two Soviet mathematicians, **Georgy Adelson-Velsky** and **Evgenii Landis**, published the first data structure that gave that guarantee. It is named after their initials: the **AVL tree**.

The idea is disarmingly simple. Do not let the tree get out of shape in the first place. After every insertion and every deletion, check whether any node has become lopsided, and if so, straighten it immediately with a local rearrangement that costs $O(1)$.

DEFINITION (balance factor). For a node x, the **balance factor** is

$$\mathrm{bf}(x) = h(\mathrm{right}(x)) - h(\mathrm{left}(x)), \tag{6.3}$$

where $h(\cdot)$ is the height of a subtree and the empty subtree has height -1.

DEFINITION (AVL tree). An **AVL tree** is a binary search tree in which

$$\mathrm{bf}(x) \in \{-1, 0, +1\} \quad \text{for every node } x. \tag{6.4}$$

Equivalently: at every node, the heights of the two subtrees differ by at most one.

In words, Equation (6.4) says: for each node, the heights of the left and right branches differ by at most 1. The sign convention of Equation (6.3) matters and is worth memorizing: the balance factor is the height of the right branch minus the height of the left branch. A negative balance factor therefore means **left-heavy**, a positive one means **right-heavy**, and zero means the two sides are the same height.

Here is one tree of each kind. **Figure 6.7** is an AVL tree; **Figure 6.8** is not.

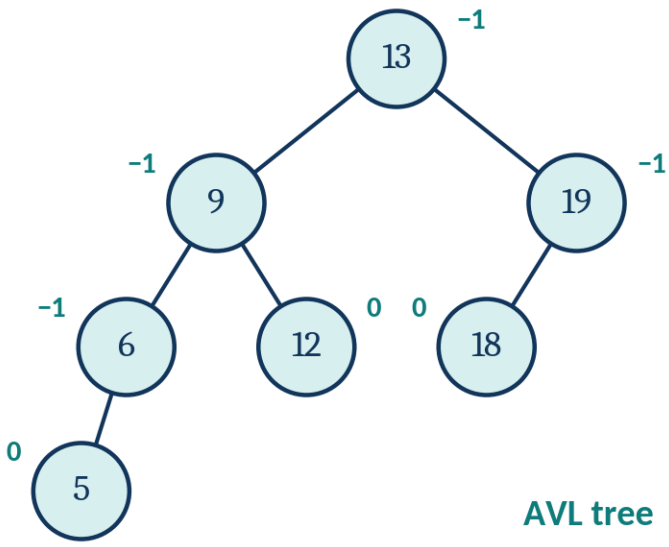


Figure 6.7 An AVL tree. Every node's two subtrees differ in height by at most one.

Check **Figure 6.7** node by node. The tree is: root 13, left child 9, right child 19; node 9 has children 6 and 12; node 6 has a left child 5; node 19 has a left child 18.

- Node 5: leaf. Both subtrees empty, heights -1 and -1. bf $= 0$. ✓
- Node 12: leaf, bf $= 0$. ✓

- Node 18: leaf, bf $= 0$. ✓
- Node 6: left subtree is the single node 5, height 0; right subtree is empty, height -1. bf $= -1 - 0 = -1$. ✓
- Node 9: left subtree is rooted at 6 with height 1; right subtree is the single node 12, height 0. bf $= 0 - 1 = -1$. ✓
- Node 19: left subtree is 18, height 0; right subtree empty, height -1. bf $= -1 - 0 = -1$. ✓
- Node 13: left subtree rooted at 9 has height 2; right subtree rooted at 19 has height 1. bf $= 1 - 2 = -1$. ✓

Every balance factor is in $\{-1, 0, +1\}$, so the tree is an AVL tree.

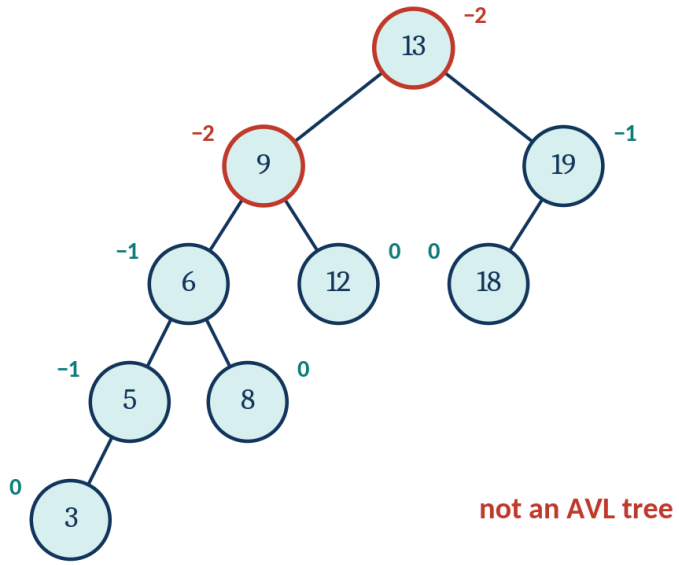


Figure 6.8 A binary search tree that is **not** an AVL tree. Adding the nodes 8 and 3 pushes two balance factors out of range: those of 9 and 13, both -2.

Figure 6.8 is the same tree with two extra nodes: node 6 now also has a right child 8, and node 5 has acquired a left child 3. Recompute:

- Node 3: leaf, height 0.
- Node 5: left subtree is 3, height 0; right subtree empty, height -1. bf $= -1$. Still fine, and node 5 now has height 1.
- Node 8: leaf, height 0.
- Node 6: left subtree rooted at 5, height 1; right subtree is 8, height 0. bf $= 0 - 1 = -1$. Fine, and node 6 now has height 2.
- Node 9: left subtree rooted at 6, height 2; right subtree is 12, height 0. bf $= 0 - 2 = \mathbf{-2}$. **Out of range.**
- Node 13: left subtree rooted at 9, now of height 3; right subtree rooted at 19, height 1. bf $= 1 - 3 = \mathbf{-2}$. **Also out of range.**

Nodes 9 and 13 both violate the invariant, so the tree is not an AVL tree — which is exactly what the label on the figure says, and why both are outlined in red there. Note that it is still a perfectly valid *binary search tree*: the inorder walk 3,5,6,8,9,12,13,18,19 is sorted. AVL-ness is an extra condition on shape, layered on top of the ordering condition on keys.

> **The invariant is about heights, not about counts.** A node whose left subtree holds 1000 nodes and whose right subtree holds 3 nodes can still be a perfectly legal AVL node, provided

the two subtrees have heights differing by at most one. AVL trees balance *depth*, not *population*. That is enough, because depth is what searching pays for.

6.15 Why an AVL Tree Is Short: the Minimum-Nodes Recurrence

The invariant looks modest — "heights differ by at most one" — so it is genuinely surprising that it forces the height down to $O(\lg n)$. We now prove it, and the proof is a recurrence of exactly the kind Chapter 3 taught us to solve.

The trick is to turn the question inside out. Instead of asking *"how tall can a tree with n nodes be?"*, ask *"how few nodes can a tree of height h have?"* — because the flimsiest possible AVL tree of a given height is the one that gives us the bound.

DEFINITION. Let $N(h)$ be the **minimum number of nodes** in any AVL tree of height h.

The base cases.

$N(0) = 1$. A tree of height 0 is a single node with no children, and one node is the fewest possible.

$N(1) = 2$. A tree of height 1 must have at least one child hanging below the root, and a root plus one child is a legal AVL tree (its balance factor is ± 1). Two nodes.

The recurrence. Take an AVL tree of height $h \geq 2$ with as few nodes as possible. Its root has two subtrees; call them A and B. Now reason about them.

1. The tree has height h, which means the longest downward path has h edges. That path goes through one of the two subtrees. So at least one subtree has height exactly $h - 1$. Say it is A: $h(A) = h - 1$.
2. What can $h(B)$ be? The AVL invariant at the root says $|h(A) - h(B)| \leq 1$, so $h(B) \in \{h - 2, h - 1, h\}$. It cannot be h, since then the tree would be taller than h. So $h(B)$ is $h - 1$ or $h - 2$.
3. We want as *few* nodes as possible, and a shorter subtree needs fewer nodes, so we choose $h(B) = h - 2$ — the smallest the invariant permits.
4. A is itself an AVL tree of height $h - 1$, and to minimize the total we make it the smallest such tree, with $N(h - 1)$ nodes. Likewise B is the smallest AVL tree of height $h - 2$, with $N(h - 2)$ nodes.
5. Add the root.

That gives, for every $h \geq 2$,

$$N(h) = N(h-1) + N(h-2) + 1, \qquad N(0) = 1, \quad N(1) = 2. \tag{6.5}$$

The first few values, computed by hand.

$N(2) = N(1) + N(0) + 1 = 2 + 1 + 1 = 4.$

$N(3) = N(2) + N(1) + 1 = 4 + 2 + 1 = 7.$

$N(4) = N(3) + N(2) + 1 = 7 + 4 + 1 = 12.$

$N(5) = N(4) + N(3) + 1 = 12 + 7 + 1 = 20.$

$N(6) = N(5) + N(4) + 1 = 20 + 12 + 1 = 33.$

So an AVL tree of height 6 must contain at least 33 nodes. Turn that around: 32 **nodes can never be arranged into an AVL tree of height** 6**.** They must fit into height 5 or less.

The trees that achieve these minima are called **Fibonacci trees**, and the name is a clue.

6.15.1 The Fibonacci connection

Recall the Fibonacci numbers, defined by $F(0) = 0$, $F(1) = 1$, and $F(m) = F(m-1) + F(m-2)$:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ 233,\ \ldots$$

The recurrence Equation (6.5) is the Fibonacci recurrence with an annoying $+1$ attached. Such a $+1$ is removed by a change of variable. Put

$$M(h) = N(h) + 1.$$

Then, substituting $N = M - 1$ into the recurrence,

$$M(h) - 1 = (M(h-1) - 1) + (M(h-2) - 1) + 1,$$

and the constants collapse:

$$M(h) = M(h-1) + M(h-2).$$

That is the Fibonacci recurrence exactly. Its base values are $M(0) = N(0) + 1 = 2$ and $M(1) = N(1) + 1 = 3$. Looking along the Fibonacci list, $2 = F(3)$ and $3 = F(4)$. Two consecutive terms agree, and both sequences obey the same recurrence, so they agree forever:

$$N(h) + 1 = F(h+3). \tag{6.6}$$

Check Equation (6.6) against the hand computations: $N(4) + 1 = 13$ and $F(7) = 13$ ✓; $N(6) + 1 = 34$ and $F(9) = 34$ ✓.

Table 6.4 lists the numbers together.

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

6.15.2 From Fibonacci to $1.44\lg n$

Chapter 3 gave the closed form of the Fibonacci numbers,

$$F(m) = \frac{\varphi^m - \hat{\varphi}^m}{\sqrt{5}}, \qquad \varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803, \qquad \hat{\varphi} = \frac{1-\sqrt{5}}{2} \approx -0.61803. \tag{6.7}$$

In Equation (6.7), because $|\hat{\varphi}| < 1$, the term $\hat{\varphi}^m$ shrinks toward zero, and in fact $|\hat{\varphi}^m|/\sqrt{5} < 1/2$ for every $m \geq 0$. So $F(m)$ is always the nearest whole number to $\varphi^m/\sqrt{5}$, and in particular

$$F(m) + 1 \;\geq\; \frac{\varphi^m}{\sqrt{5}}. \tag{6.8}$$

Now let T be any AVL tree with n nodes and height h. Since $N(h)$ is the *fewest* nodes such a tree can have,

$$n \;\geq\; N(h) \;=\; F(h+3) - 1.$$

Add 2 to both sides and apply Equation (6.8) with $m = h + 3$:

$$n + 2 \;\geq\; F(h+3) + 1 \;\geq\; \frac{\varphi^{h+3}}{\sqrt{5}}.$$

Take base-2 logarithms of both sides — legal because both sides are positive, and lg is increasing:

$$\lg(n+2) \;\geq\; (h+3)\lg\varphi - \frac{1}{2}\lg 5.$$

Rearrange for h, dividing by the positive constant $\lg\varphi$:

$$h + 3 \;\leq\; \frac{\lg(n+2) + 1/2\lg 5}{\lg\varphi}.$$

Put in the numbers: $\lg\varphi = \lg 1.61803 \approx 0.69424$, so $1/\lg\varphi \approx 1.44042$; and $1/2\lg 5 \approx 1.16096$, so $1/2\lg 5 \,/\, \lg\varphi \approx 1.67231$. Hence

$$h \;\leq\; 1.44042\lg(n+2) + 1.67231 - 3,$$

that is,

$$\boxed{h \;\leq\; 1.4405\lg(n+2) - 1.3277} \tag{6.9}$$

> **What the bound says in words.** An AVL tree is never more than about **44% taller** than a perfectly balanced binary tree of the same size. Since Equation (6.9) gives $h \leq 1.4405\lg(n+2) - 1.3277 = O(\lg n)$, every operation that costs $O(h)$ — search, minimum, maximum, successor, predecessor, insert, delete — costs $O(\lg n)$ in an AVL tree. Not on average. Not usually. Always, on every input, in every order.

A note on the constant you will see elsewhere. The bound is very often quoted as $h < 1.4405\lg(n+2) - 0.3277$, exactly one larger than ours. That version counts *nodes* on the longest path rather than *edges*, so its h is one bigger than ours for the same tree. Both statements are correct; they differ only in the convention fixed back in §6.2. Always check which one a source is using before comparing constants.

Listing 6.6 computes the table, verifies $N(h) + 1 = F(h+3)$ numerically, checks the bound at every height, and then answers the practical question in the other direction: given n, how tall can the tree possibly get?

Listing 6.6 The minimum-nodes recurrence, the Fibonacci identity and the height bound, all checked numerically. Compile with `g++ -std=c++17 -O2 -o minnodes minnodes.cpp`.

```
// The smallest AVL tree of each height, the Fibonacci numbers hiding inside it,
// and the bound h < 1.4405 * lg(n+2) - 1.3277 checked against the real numbers.
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>

int main() {
    const int H = 45;
    std::vector<long long> N(H + 1), F(H + 4);

    N[0] = 1;                        // a single node is an AVL tree of height 0
    N[1] = 2;                        // a root with one child has height 1
    for (int h = 2; h <= H; ++h) N[h] = N[h - 1] + N[h - 2] + 1;

    F[0] = 0; F[1] = 1;
    for (int i = 2; i < (int)F.size(); ++i) F[i] = F[i - 1] + F[i - 2];

    std::cout << " h        N(h)   N(h)+1   F(h+3)   bound 1.4405*lg(N(h)+2)-1.3277\n";
    std::cout << " ------------------------------------------------------------\n";
    for (int h = 0; h <= 12; ++h) {
        double bound = 1.4405 * std::log2((double)N[h] + 2.0) - 1.3277;
        std::cout << std::setw(2) << h
                  << std::setw(12) << N[h]
                  << std::setw(9)  << N[h] + 1
                  << std::setw(9)  << F[h + 3]
                  << std::setw(23) << std::fixed << std::setprecision(4) << bound
                  << (bound > h ? "   ok" : "   FAILS") << "\n";
    }

    std::cout << "\nHeight of the tallest AVL tree that can hold n nodes:\n";
    for (long long n : {1000LL, 1000000LL, 1000000000LL}) {
        int h = 0;
        while (h + 1 <= H && N[h + 1] <= n) ++h;
        std::cout << "  n = " << std::setw(11) << n
                  << "   tallest AVL height = " << std::setw(3) << h
                  << "   lg n = " << std::fixed << std::setprecision(2)
                  << std::log2((double)n)
                  << "   1.44*lg n = " << 1.4405 * std::log2((double)n) << "\n";
    }
    return 0;
}
```

The program prints:

```
 h        N(h)   N(h)+1   F(h+3)   bound 1.4405*lg(N(h)+2)-1.3277
 ------------------------------------------------------------
 0          1        2        2                 0.9554   ok
 1          2        3        3                 1.5533   ok
 2          4        5        5                 2.3959   ok
 3          7        8        8                 3.2386   ok
 4         12       13       13                 4.1568   ok
 5         20       21       21                 5.0961   ok
 6         33       34       34                 6.0610   ok
 7         54       55       55                 7.0378   ok
 8         88       89       89                 8.0238   ok
 9        143      144      144                 9.0150   ok
10        232      233      233                10.0096   ok
11        376      377      377                11.0062   ok
12        609      610      610                12.0042   ok

Height of the tallest AVL tree that can hold n nodes:
  n =        1000   tallest AVL height =  13   lg n = 9.97   1.44*lg n = 14.36
  n =     1000000   tallest AVL height =  27   lg n = 19.93   1.44*lg n = 28.71
  n =  1000000000   tallest AVL height =  41   lg n = 29.90   1.44*lg n = 43.07
```

Three things to take away from that output. First, the column $N(h) + 1$ and the column $F(h + 3)$ are identical, which is the identity we proved. Second, the bound holds at every height and gets *tighter* as h grows — at $h = 12$ it gives 12.0042, a margin of four thousandths. That is the sign of a bound that is essentially exact, not a generous over-estimate. Third, and most practically: a billion keys in an AVL tree can never be more than 41 levels deep. A search touches at most 42 nodes. Compare that with the plain binary search tree of §6.13, which on sorted input would have been a billion levels deep.

6.16 Rotations: the One Tool

We now know what we want (balance) and why we want it (height $O(\lg n)$). We need a way to *restore* balance after an insertion or deletion has broken it, and the repair must be cheap — $O(1)$ — or we will have paid for the guarantee with a slower operation.

That tool is the **rotation**, pictured in **Figure 6.9**. It serves insertion and deletion alike.

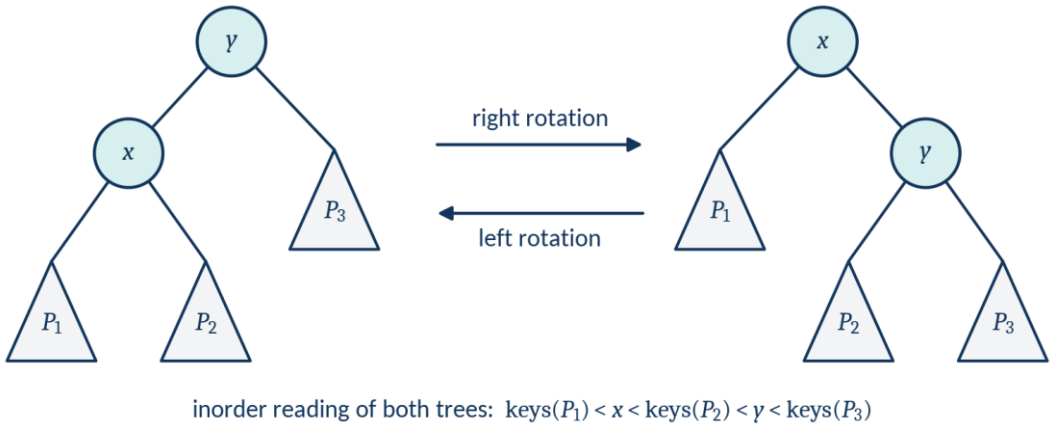


Figure 6.9 The two rotations. The same five items — the nodes x and y and the subtrees P_1, P_2, P_3 — appear in both pictures. A right rotation turns the left picture into the right one; a left rotation turns the right picture back into the left one. The two are exact inverses.

Look at the left-hand picture. The node y is on top; its left child is x; its right subtree is P_3. Node x has subtrees P_1 on the left and P_2 on the right.

Because this is a binary search tree, the BST property at y and at x together pin down the order of everything:

$$\text{keys}(P_1) \;<\; x \;<\; \text{keys}(P_2) \;<\; y \;<\; \text{keys}(P_3). \tag{6.10}$$

Read Equation (6.10) left to right: P_1, then the single key x, then P_2, then the single key y, then P_3. That is precisely the inorder walk of the left-hand tree.

Now look at the right-hand picture. The node x is on top; its left subtree is P_1; its right child is y; and y has P_2 on the left and P_3 on the right. Read *its* inorder walk: P_1, x, P_2, y, P_3.

The two inorder walks are identical. That is the whole point of a rotation, and it is worth stating as a theorem.

> **THEOREM (rotations preserve the BST property).** A rotation changes the *shape* of a tree without changing its *inorder sequence*. Since a binary tree is a binary search tree exactly when its inorder walk is increasing (§6.6.1), and a rotation leaves the inorder walk untouched, a rotation performed on a binary search tree always yields a binary search tree.

6.16.1 Right rotation, with the exact pointer updates

A **right rotation about** y takes the left-hand picture of **Figure 6.9** to the right-hand one. In words: x moves up and becomes the new root of this subtree; y moves down and becomes x's right child; and the subtree P_2, which was x's right subtree, is handed over to become y's left subtree.

Only P_2 changes parent. P_1 stays with x; P_3 stays with y. **Listing 6.7** writes the pointer assignments in the order in which they must be performed, inside a complete program that stores a height in every node and rotates a small left-leaning chain.

Listing 6.7 A right rotation with its exact pointer updates, applied to the left-leaning chain 30 → 20 → 10. Compile with `g++ -std=c++17 -O2 -o rotright rotright.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;
}
```

Each node is printed as `key[height]`. The program prints:

```
before: 30[2]{20[1]{10[0],.},.}
after : 20[1]{10[0],30[0]}
```

Before the rotation the chain has height 2; afterward 20 sits on top with 10 and 30 as its children, and the height is 1.

Steps 5 and 6 inside `rotateRight` must be in that order. The height of x is computed from the heights of its children, and one of its children is now y; so y's height must already be correct when x is refreshed. Reversing those two lines is the single most common bug in a hand-written AVL tree, and it produces a tree that looks right and silently stores wrong heights.

The whole procedure touches a constant number of pointers, so a rotation costs $\Theta(1)$ — independent of how big P_1, P_2 and P_3 are. Notice that the subtrees are never traversed, only re-pointed.

6.16.2 Left rotation

A **left rotation about** x is the mirror image: it takes the right-hand picture of **Figure 6.9** back to the left-hand one. Now y moves up, x moves down to become y's left child, and P_2 — this time y's left subtree — is handed to x as its right subtree.

Listing 6.8 is the complete program for this direction; its `main` performs exactly the rotation worked by hand in the example that follows.

Listing 6.8 A left rotation, applied to the degenerate chain 10 → 20 → 30. Compile with `g++ -std=c++17 -O2 -o rotleft rotleft.cpp`.

```
// A left rotation, the mirror image, applied to the chain 10 -> 20 -> 30.
#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* rotateLeft(Node* x) {
    Node* y  = x->right;        // the node that will move up
    Node* P2 = y->left;         // the subtree that will change parent
    y->left  = x;
    x->right = P2;
    refresh(x);                 // x is now lower, so refresh it first
    refresh(y);
    return y;
}

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;
}

void inorder(const Node* x) {
    if (!x) return;
    inorder(x->left); std::cout << x->key << " "; inorder(x->right);
}
```

Listing 6.8 (Continued) *A left rotation, applied to the degenerate chain 10 \to 20 \to 30. Compile with g++ -std=c++17 -O2 -o rotleft rotleft.cpp.*

```
int main() {
    Node* a = new Node(10); Node* b = new Node(20); Node* c = new Node(30);
    a->right = b; b->right = c;                  // the degenerate chain 10 -> 20 -> 30
    refresh(c); refresh(b); refresh(a);
    std::cout << "before: " << shape(a) << "   inorder: "; inorder(a); std::cout << "\n";
    Node* root = rotateLeft(a);
    std::cout << "after : " << shape(root) << "   inorder: "; inorder(root); std::cout <<
"\n";
    delete a; delete b; delete c;
    return 0;
}
```

It prints:

```
before: 10[2]{.,20[1]{.,30[0]}}   inorder: 10 20 30
after : 20[1]{10[0],30[0]}   inorder: 10 20 30
```

A rotation by hand. Take the three-node chain $10 \to 20 \to 30$, where 10 is the root, 20 is its right child and 30 is 20's right child. Its height is 2; it is a degenerate tree; and its inorder walk is 10,20,30.

Perform a left rotation about the root, $x = 10$. Then $y = x$->right $= 20$, and $P_2 = y$->left, which is empty. Now y->left $= x$ makes 10 the left child of 20; and x->right $= P_2$ makes 10's right subtree empty. The new root is 20, with children 10 and 30.

The new tree has height 1, one less than before, and its inorder walk is still 10,20,30. One rotation, three pointer writes, and the tree went from degenerate to perfectly balanced.

6.16.3 What a rotation does to the height

This is the fact that makes the whole scheme work. Look again at **Figure 6.9**. In the left picture the path from the root y down into P_1 passes through y and x: depth 2 above P_1. In the right picture the path from the root x down into P_1 passes only through x: depth 1.

So a right rotation **lifts P_1 one level** and **lowers P_3 one level**, while P_2 stays where it is. A left rotation does the reverse. A rotation therefore moves height from one side of a subtree to the other, one level at a time — which is exactly the correction needed when one side has become one level too tall.

6.17 The Four Rotation Cases

Now we put rotations to work. Suppose we have just inserted a new node into an AVL tree using the ordinary BST insertion of §6.10. The new node is a leaf, so its own balance factor is 0; but the subtrees of every node on the path from the root down to it have just grown by at most one level, so some of those balance factors may have moved out of range.

The procedure has four steps.

AVL INSERTION. Let s be the node being inserted.

1. Perform the standard binary-search-tree insertion of s.
2. Starting at s, walk **upward** and find the first node whose balance factor has left $\{-1, 0, +1\}$. Call that node z — it is the deepest unbalanced node.
3. Let y be the child of z that lies on the path from s up to z, and let x be the grandchild of z on that same path.

4. Rebalance the subtree rooted at z with the rotation or rotations that its case requires. There are exactly four cases, because x can sit in four positions relative to y and z:
   - y is the **left** child of z and x is the **left** child of y — the **Left-Left (LL)** case;
   - y is the **left** child of z and x is the **right** child of y — the **Left-Right (LR)** case;
   - y is the **right** child of z and x is the **right** child of y — the **Right-Right (RR)** case;
   - y is the **right** child of z and x is the **left** child of y — the **Right-Left (RL)** case.

The names read from the top down: the first letter says which way y hangs off z, the second says which way x hangs off y. "Left-Left" means the tree has leaned left twice in a row.

In all four figures that follow, P_1, P_2, P_3, P_4 are subtrees, drawn as triangles because their contents do not matter — only their heights do. Assume throughout the following heights, measured *after* the insertion. The two subtrees hanging below x have heights k and $k-1$: the one of height k is the one that received the new node (it had height $k-1$ before), and its sibling did not change. The other two subtrees have height k. Before the insertion, then, every node was balanced; the insertion lifted x from height k to $k+1$, and that extra level is what travels up to z.

6.17.1 Case LL — one right rotation

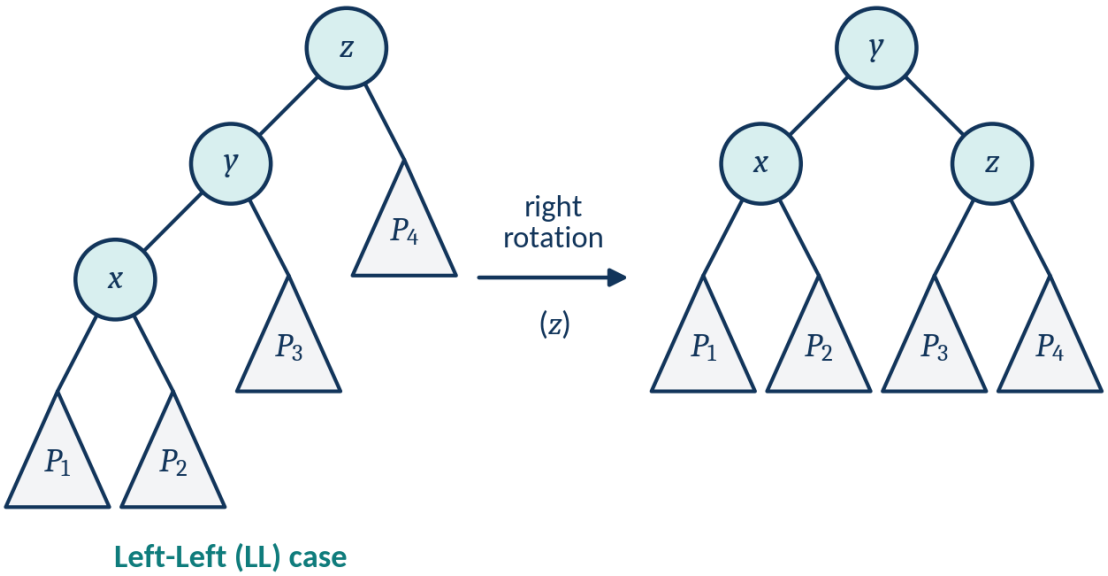


Figure 6.10 The Left-Left case, which also arises during deletion. A single right rotation about z restores the balance; y becomes the new root of the subtree, with x and z as its children.

The situation. In the left half of **Figure 6.10**, z is on top with right subtree P_4; y is z's left child, with right subtree P_3; x is y's left child, with subtrees P_1 and P_2.

The heights. P_1 (which received the new node), P_3 and P_4 have height k, and P_2 has height $k-1$; x has height $k+1$; y has height $\max(k+1, k)+1 = k+2$; z has height $\max(k+2, k)+1 = k+3$. Therefore

$$\mathrm{bf}(z) = h(P_4) - h(y) = k - (k+2) = -2,$$

which is out of range: z is left-heavy by two. And

$$\mathrm{bf}(y) = h(P_3) - h(x) = k - (k+1) = -1,$$

which is negative — left-heavy. Two leftward leans in a row: this is LL.

The fix. One right rotation about z. **Pseudocode 6.1** is the code of §6.16.1 with the roles renamed, $y \mapsto z$ and $x \mapsto y$:

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
```

The result. The right half of **Figure 6.10**: y on top, with left child x (holding P_1, P_2) and right child z (holding P_3, P_4).

Check the heights afterward. x still has height $k+1$ and balance factor -1. z now has children P_3 and P_4, both of height k, so $h(z) = k+1$ and $\mathrm{bf}(z) = 0$. And y has two children of height $k+1$, so $h(y) = k+2$ and $\mathrm{bf}(y) = 0$. Every balance factor is legal.

And the subtree is back to its pre-insertion height. Before the insertion, the subtree rooted at z had height $k+2$; the insertion pushed it to $k+3$; after the rotation it is $k+2$ again. This is the crucial observation of the whole section, and §6.19.2 will draw the consequence from it.

6.17.2 Case LR — a left rotation then a right rotation

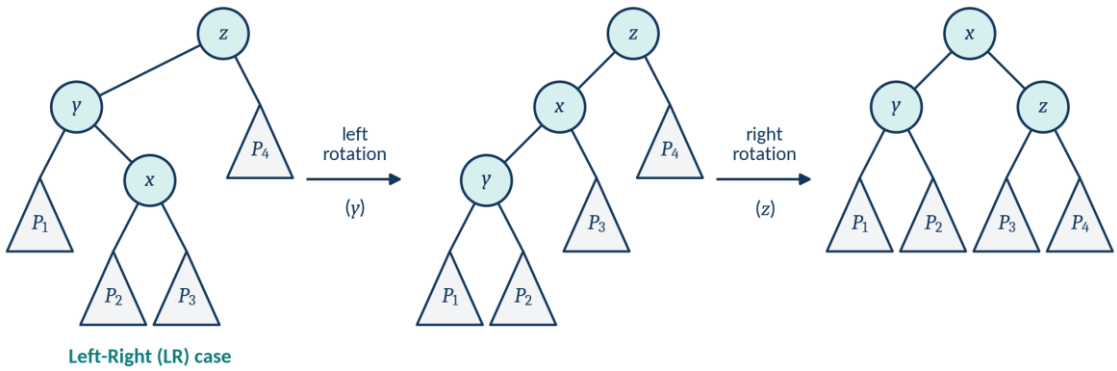


Figure 6.11 The Left-Right case, which also arises during deletion. Two rotations are needed: first a left rotation about y, which converts the problem into an LL case, then a right rotation about z.

The situation. In the left third of **Figure 6.11**, z is on top with right subtree P_4; y is z's left child, with *left* subtree P_1; x is y's *right* child, with subtrees P_2 and P_3.

The heights. P_1 and P_4 have height k; x has height $k+1$ (its taller child has height k); y has height $k+2$; z has height $k+3$. So

$$\mathrm{bf}(z) = k - (k+2) = -2 \quad \text{(left-heavy, out of range)},$$

$$\mathrm{bf}(y) = h(x) - h(P_1) = (k+1) - k = +1 \quad \text{(right-heavy)}.$$

Leaning left, then right: this is LR.

Why one rotation is not enough. Try a single right rotation about z. It would lift y to the top, give y the subtree P_1 on the left and z on the right, and hand x — which is y's right subtree — over to z as its left subtree. Then z would have x (height $k+1$) on the left and P_4 (height k) on the right, so $h(z) = k+2$; while y's left child P_1 has height k. We would get $\mathrm{bf}(y) = (k+2) - k = +2$: still broken, just leaning the other way. A single rotation cannot fix a zig-zag.

The fix, in two steps.

Step 1: left rotation about y. This is the middle picture of **Figure 6.11**. Node x moves up to become z's left child; y moves down to become x's left child; and P_2, which was x's left subtree, is handed to y as its right subtree. The chain $z \to y \to x$ zig-zagged left-then-right; now the chain is $z \to x \to y$, which leans left twice. The problem has been converted into an LL case.

Step 2: right rotation about z. Exactly as in §6.17.1. Node x moves to the top; y and z become its two children. **Pseudocode 6.2** performs both steps.

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
```

The result. The right third of **Figure 6.11**: x on top; left child y holding P_1 and P_2; right child z holding P_3 and P_4.

Check the heights. y has children P_1 and P_2; z has children P_3 and P_4. Each of those four subtrees has height k or $k-1$ — one of P_2, P_3 grew by the insertion, but neither exceeds k. So $h(y) = h(z) = k+1$ at most, and $h(x) = k+2$, with all three balance factors in $\{-1, 0, +1\}$. The subtree is again $k+2$ tall, its pre-insertion height.

6.17.3 Case RR — one left rotation

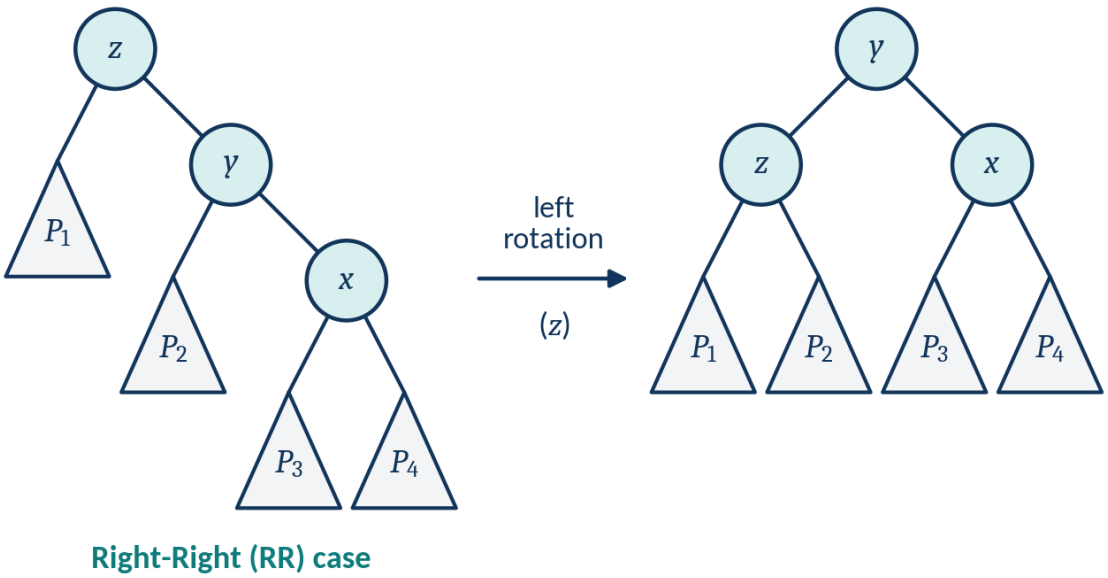


Figure 6.12 The Right-Right case, which also arises during deletion. The exact mirror image of **Figure 6.10**: a single left rotation about z.

The situation. In the left half of **Figure 6.12**, z is on top with left subtree P_1; y is z's right child with left subtree P_2; x is y's right child with subtrees P_3 and P_4.

The heights. $h(x) = k+1, h(y) = k+2, h(z) = k+3$, so

$$\mathrm{bf}(z) = h(y) - h(P_1) = (k+2) - k = +2 \quad \text{(right-heavy, out of range)},$$

$$\mathrm{bf}(y) = h(x) - h(P_2) = (k+1) - k = +1 \quad \text{(right-heavy)}.$$

Two rightward leans: RR.

The fix. One left rotation about z, in **Pseudocode 6.3**:

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);
```

The result. The right half of **Figure 6.12**: y on top, left child z (holding P_1, P_2), right child x (holding P_3, P_4). Nodes z and y now have balance factor 0, x keeps the ± 1 it already had, and the subtree is back to height $k + 2$.

6.17.4 Case RL — a right rotation then a left rotation

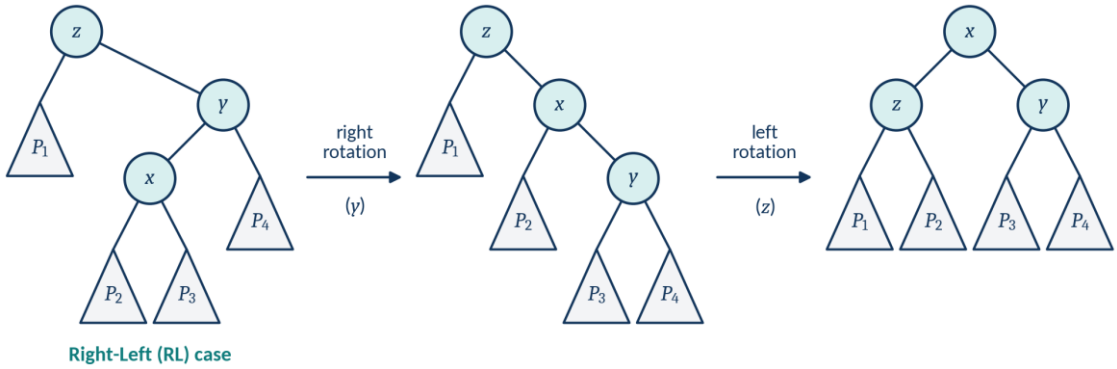


Figure 6.13 The Right-Left case, which also arises during deletion. The mirror of **Figure 6.11**: first a right rotation about y, which turns the zig-zag into an RR case, then a left rotation about z.

The situation. In the left third of **Figure 6.13**, z has left subtree P_1; y is z's right child with right subtree P_4; x is y's *left* child, with subtrees P_2 and P_3.

The heights. $\mathrm{bf}(z) = +2$ and $\mathrm{bf}(y) = h(P_4) - h(x) = k - (k+1) = -1$. Leaning right, then left: RL.

The fix, in two steps. A right rotation about y pulls x up into y's place and pushes y down to the right — the middle picture — turning the zig-zag into a double right lean. Then a left rotation about z finishes the job, as **Pseudocode 6.4** shows.

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
```

The result. The right third of **Figure 6.13**: x on top; left child z holding P_1 and P_2; right child y holding P_3 and P_4. Height $k + 2$ again.

6.17.5 Telling the four cases apart in code

The decision rule needs only two balance factors: the one at z and the one at y.

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**

The same rule can be phrased as a comparison of signs along the path $z \to y \to x \to s$: for LL the signs go $(-)$ to $(-)$, for LR they go $(-)$ to $(+)$, for RR $(+)$ to $(+)$, and for RL $(+)$ to $(-)$. Same rule, said out loud.

> **What if** $\mathrm{bf}(y) = 0$**?** After an *insertion* this cannot happen: the subtree rooted at y grew, so one of its sides is strictly taller. After a *deletion* it can and does happen, and then the single rotation is the one that restores the invariant. The universal convention — and the one used in the code of §6.19 — is to treat $\mathrm{bf}(y) = 0$ as the single-rotation case, that is, to write `if (bf(z->left) <= 0) rotateRight(z);` rather than `< 0`. Getting this wrong and performing the double rotation can leave y with a balance factor of ± 2: the tree is still correct as a search tree, but it is no longer an AVL tree.

6.18 A Worked Insertion Example, Step by Step

Figure 6.14 to **Figure 6.18** work through five insertions, one per figure, and between them they exercise every case in **Table 6.5**. We reproduce them here one at a time, with the balance factors computed by hand and confirmed by **Listing 6.9** in §6.19.

Throughout, a balance factor is written beside its node in these figures, in teal, using the same convention as ours: right height minus left height.

6.18.1 Step 1 — insert 14: no rebalancing needed

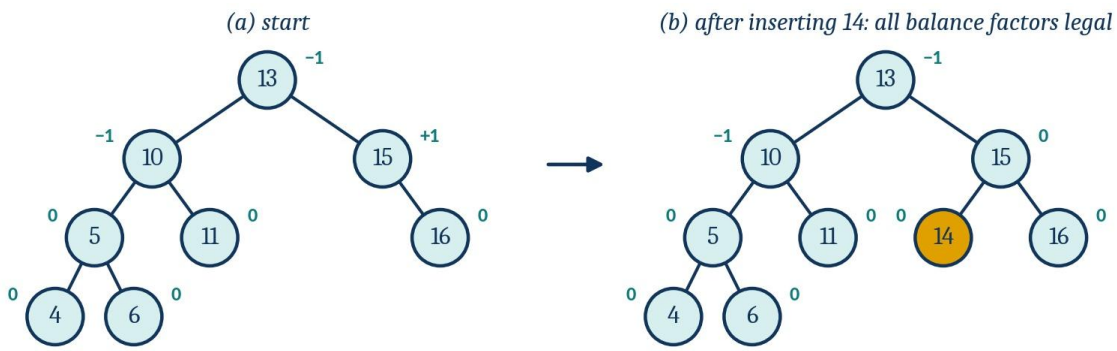


Figure 6.14 Inserting 14 into an AVL tree. Every balance factor stays inside $\{-1, 0, +1\}$, so no rotation is performed.

The tree on the left of **Figure 6.14** is: root 13; left child 10 with children 5 and 11; node 5 with children 4 and 6; right child 15 with a right child 16.

Insert 14. The descent compares 14 against 13 — greater, go right — then against 15 — smaller, go left — and finds an empty slot. Node 14 becomes the left child of 15.

Now recompute balance factors upward from the new node.

- Node 14 is a leaf: bf $= 0$, height 0.
- Node 15 now has left child 14 (height 0) and right child 16 (height 0). bf(15) $= 0 - 0 = 0$, height 1. Before the insertion it was $+1$; the insertion *improved* it.
- Node 13 has left subtree rooted at 10, of height 2, and right subtree rooted at 15, of height 1. bf(13) $= 1 - 2 = -1$, height 3.

No balance factor left the legal range, so there is nothing to repair. This is the common case: most insertions into an AVL tree require no rotation at all.

6.18.2 Step 2 — insert 3: the Left-Left case

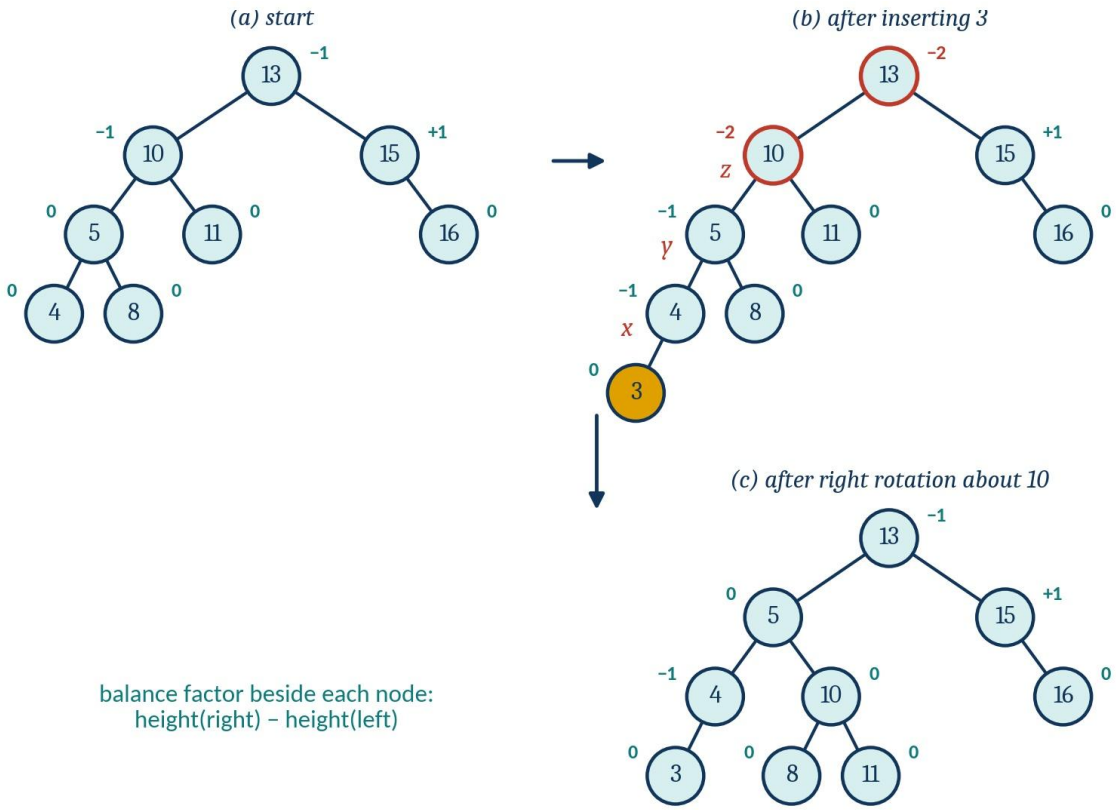


Figure 6.15 Inserting 3 produces a balance factor of -2 at node 10. The case is Left-Left, and one right rotation with 10 as the pivot repairs it.

The starting tree in **Figure 6.15** is: root 13; left child 10 with children 5 and 11; node 5 with children 4 and 8; right child 15 with right child 16.

Insert 3: at 13 go left, at 10 go left, at 5 go left, at 4 go left into an empty slot. Node 3 becomes the left child of 4.

Walk back up, recomputing.

- Node 3: leaf, height 0, bf $= 0$.
- Node 4: left child 3 (height 0), right empty (height -1). bf(4) $= -1 - 0 = -1$, height 1. Legal.
- Node 5: left subtree rooted at 4 (height 1), right child 8 (height 0). bf(5) $= 0 - 1 = -1$, height 2. Legal.
- Node 10: left subtree rooted at 5 (height 2), right child 11 (height 0). bf(10) $= 0 - 2 = \mathbf{-2}$. **Illegal.** Stop here: $z = 10$.

Identify the three nodes. $z = 10$. The path from the new node 3 up to z runs $3 \to 4 \to 5 \to 10$, so the child of z on that path is $y = 5$ and the grandchild is $x = 4$. Node y is the *left* child of z; node x is the *left* child of y. Left-Left.

Cross-check with **Table 6.5**: $\text{bf}(z) = -2 < -1$, and $\text{bf}(y) = \text{bf}(5) = -1$, negative. LL confirmed.

Apply one right rotation about $z = 10$, exactly as **Figure 6.10** prescribes. Here $x = 5$ in the rotation code's naming (the left child of the pivot), and the subtree that changes parent is 5's right child, node 8.

1. $x \leftarrow$ 10->left $= 5$.
2. $P_2 \leftarrow$ 5->right $= 8$.
3. 5->right $\leftarrow 10$.
4. 10->left $\leftarrow 8$.
5. Refresh 10, then 5.
6. The new subtree root, 5, is attached where 10 used to be: as the left child of 13.

The result is the bottom-right tree of **Figure 6.15**: root 13; left child 5 with children 4 (holding 3) and 10 (holding 8 and 11); right child 15 with right child 16.

Verify: $\text{bf}(4) = -1$; $\text{bf}(10) = 0$; $\text{bf}(5) = h(10) - h(4) = 1 - 1 = 0$; $\text{bf}(13) = h(15) - h(5) = 1 - 2 = -1$. All legal. The height of the whole tree is 3, the same as before the insertion — which is why no further rotation is needed higher up.

6.18.3 Step 3 — insert 45: the Right-Right case

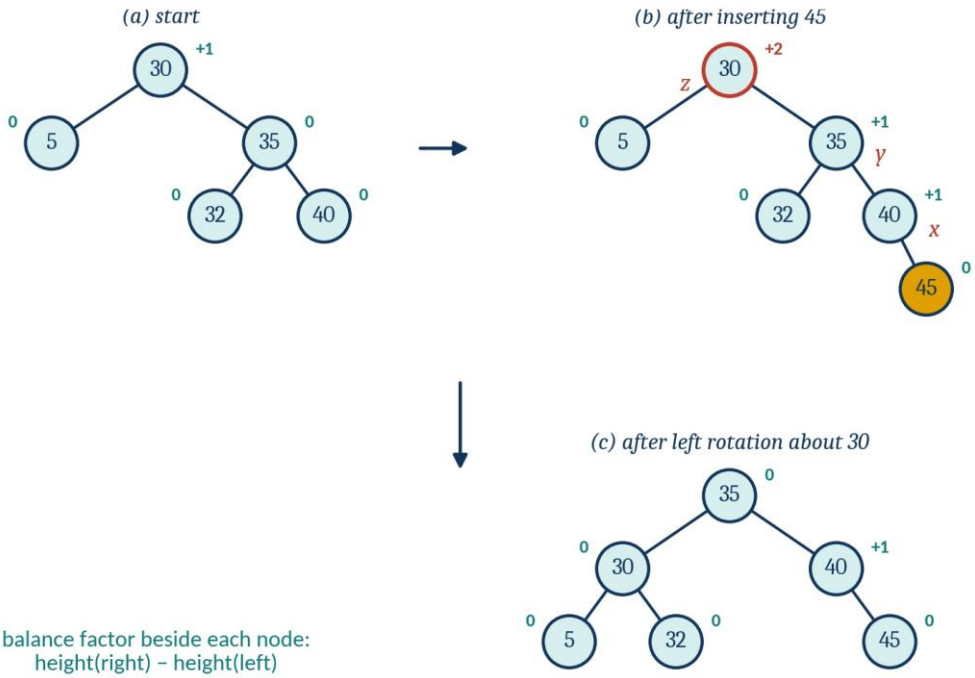


Figure 6.16 Inserting 45 makes node 30 right-heavy by two. The case is Right-Right, and one left rotation with 30 as the pivot repairs it.

The starting tree in **Figure 6.16** is: root 30, left child 5, right child 35 with children 32 and 40.

Insert 45: at 30 go right, at 35 go right, at 40 go right into an empty slot. Node 45 becomes the right child of 40.

Walking up:

- Node 45: leaf, height 0, bf = 0.
- Node 40: left empty (height −1), right child 45 (height 0). bf(40) = 0 − (−1) = + 1, height 1. Legal.
- Node 35: left child 32 (height 0), right subtree rooted at 40 (height 1). bf(35) = 1 − 0 = + 1, height 2. Legal.
- Node 30: left child 5 (height 0), right subtree rooted at 35 (height 2). bf(30) = 2 − 0 = + **2**. **Illegal.** So $z = 30$.

The path from 45 up is 45 → 40 → 35 → 30, so $y = 35$ and $x = 40$. Node y is the right child of z; node x is the right child of y. Right-Right — and bf(y) = + 1 is positive, matching the RR row of **Table 6.5**.

Apply one left rotation about $z = 30$:

1. $y \leftarrow$ 30->right = 35.
2. $P_2 \leftarrow$ 35->left = 32.
3. 35->left ← 30.
4. 30->right ← 32.
5. Refresh 30, then 35.

The result is the bottom-right tree of **Figure 6.16**: root 35; left child 30 with children 5 and 32; right child 40 with right child 45. Balance factors: bf(30) = 0, bf(40) = + 1, bf(35) = $h(40) - h(30) = 1 - 1 = 0$. All legal.

6.18.4 Step 4 — insert 7: the Left-Right case

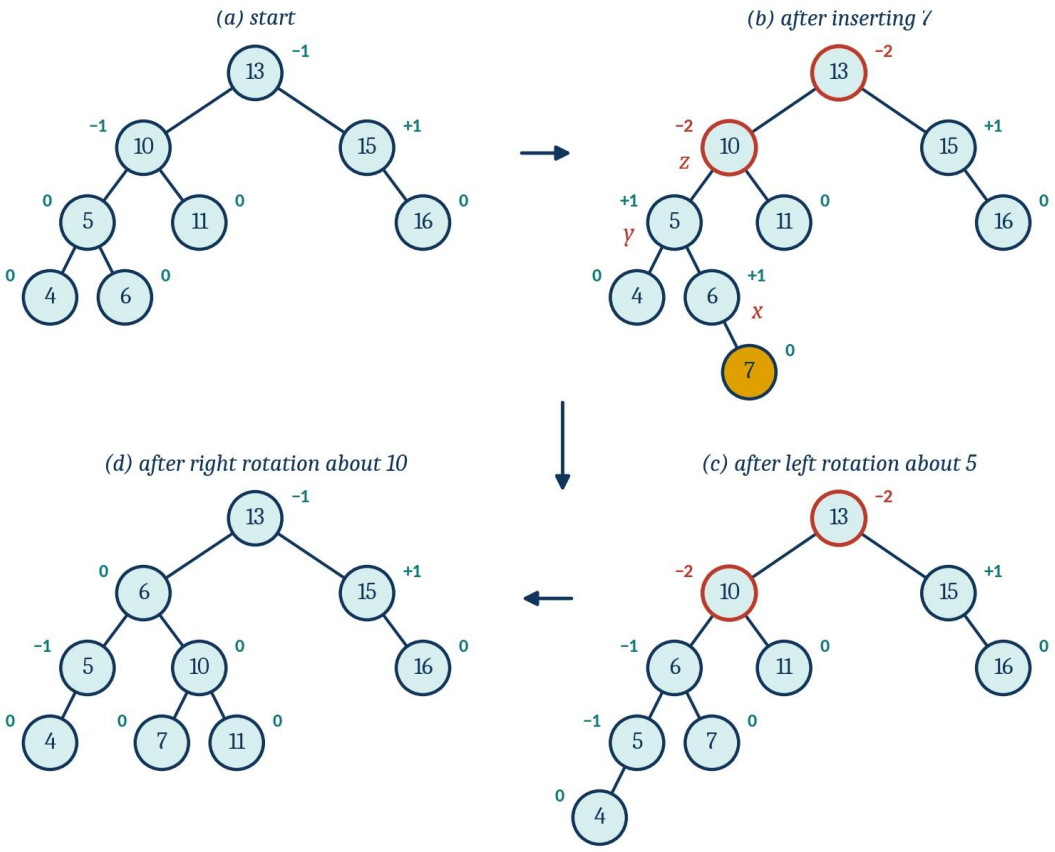


Figure 6.17 Inserting 7 creates a zig-zag — left at 10, then right at 5. Two rotations are needed: left with 5 as pivot, then right with 10 as pivot.

The starting tree is the same as in step 1: root 13; left child 10 with children 5 and 11; node 5 with children 4 and 6; right child 15 with right child 16.

Insert 7: at 13 go left, at 10 go left, at 5 go right, at 6 go right into an empty slot. Node 7 becomes the right child of 6.

Walking up:

- Node 7: leaf, height 0.
- Node 6: right child 7, left empty. $\text{bf}(6) = 0 - (-1) = +1$, height 1. Legal.
- Node 5: left child 4 (height 0), right subtree rooted at 6 (height 1). $\text{bf}(5) = 1 - 0 = +1$, height 2. Legal.
- Node 10: left subtree rooted at 5 (height 2), right child 11 (height 0). $\text{bf}(10) = 0 - 2 = \mathbf{-2}$. **Illegal.** So $z = 10$.

The path from 7 up is $7 \to 6 \to 5 \to 10$, so $y = 5$ and $x = 6$. Node y is the *left* child of z; node x is the *right* child of y. Left-Right — and $\text{bf}(y) = +1$ is positive, matching the LR row of **Table 6.5**.

Step 1: left rotation about $y = 5$. Node 6 moves up into 5's place as the left child of 10; node 5 becomes 6's left child; 6's old left subtree — empty here — goes to 5 as its right subtree. The left spine of 10 now reads $10 \to 6 \to 5 \to 4$, a double left lean. This is panel (c) of **Figure 6.17**, in which node 10 still shows -2 and the chain below it is now 6 then 5 then 4.

Step 2: right rotation about $z = 10$. Node 6 moves to the top of this subtree; node 10 becomes 6's right child; 6's old right subtree, the single node 7, is handed to 10 as its left subtree.

The result is the bottom-left tree of **Figure 6.17**: root 13; left child 6 with children 5 (holding 4) and 10 (holding 7 and 11); right child 15 with right child 16. Balance factors: $\text{bf}(5) = -1$, $\text{bf}(10) = 0$, $\text{bf}(6) = h(10) - h(5) = 1 - 1 = 0$, $\text{bf}(13) = 1 - 2 = -1$. All legal.

6.18.5 Step 5 — insert 15: the Right-Left case

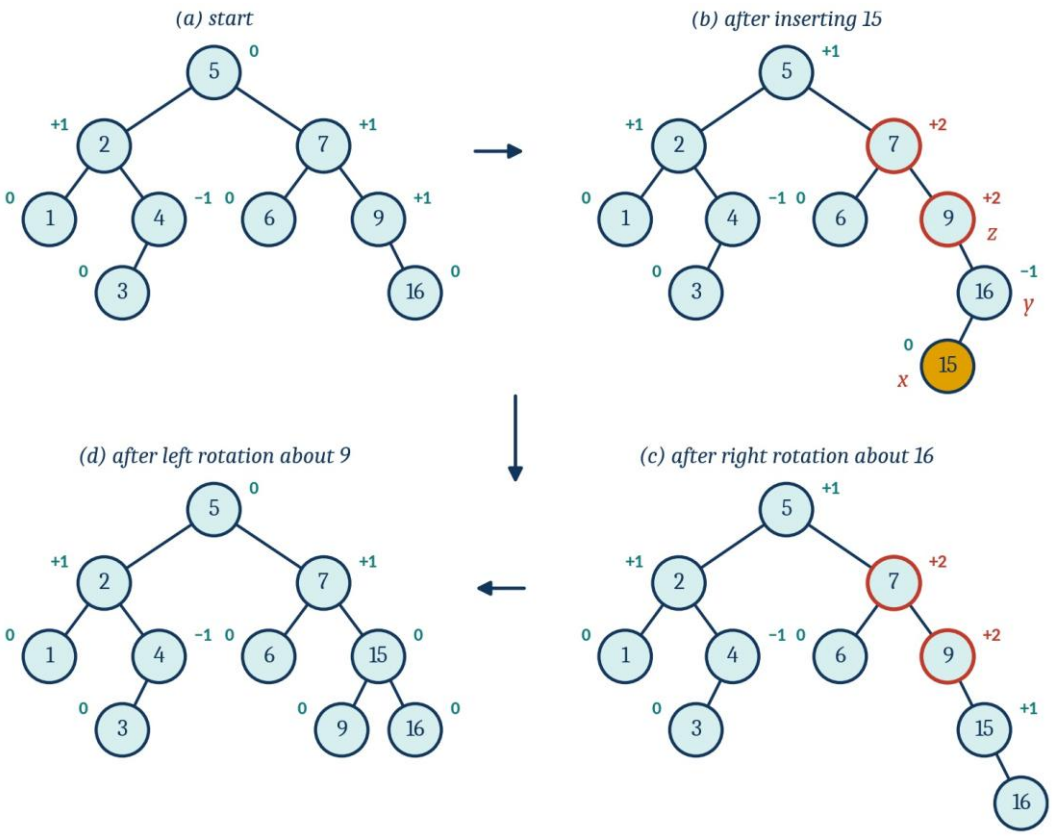


Figure 6.18 Inserting 15 makes node 9 right-heavy by two while node 16 leans left. The case is Right-Left: a right rotation with 16 as pivot, then a left rotation with 9 as pivot.

The starting tree in **Figure 6.18** is: root 5; left child 2 with children 1 and 4, and node 4 has a left child 3; right child 7 with children 6 and 9, and node 9 has a right child 16.

Insert 15: at 5 go right, at 7 go right, at 9 go right, at 16 go left into an empty slot. Node 15 becomes the left child of 16.

Walking up:

- Node 15: leaf, height 0.
- Node 16: left child 15 (height 0), right empty. $\text{bf}(16) = -1 - 0 = -1$, height 1. Legal.
- Node 9: left empty (height -1), right subtree rooted at 16 (height 1). $\text{bf}(9) = 1 - (-1) = +\mathbf{2}$. **Illegal.** So $z = 9$.

The path from 15 up is $15 \to 16 \to 9$, so $y = 16$ and $x = 15$. Node y is the *right* child of z; node x is the *left* child of y. Right-Left — and $\text{bf}(y) = -1$ is negative, matching the RL row of **Table 6.5**.

Step 1: right rotation about $y = 16$. Node 15 moves up to become 9's right child; node 16 becomes 15's right child. The chain below 9 now reads $9 \to 15 \to 16$, a double right lean.

Step 2: left rotation about $z = 9$. Node 15 moves up into 9's place as the right child of 7; node 9 becomes 15's left child.

The result is the bottom-left tree of **Figure 6.18**: root 5; left child 2 unchanged; right child 7 with left child 6 and right child 15, and 15 has children 9 and 16. Balance factors: $\text{bf}(9) = 0$, $\text{bf}(16) = 0$, $\text{bf}(15) = 0$, $\text{bf}(7) = h(15) - h(6) = 1 - 0 = +1$, $\text{bf}(5) = h(7) - h(2) = 2 - 2 = 0$. All legal.

Five insertions, four cases, and every one of them repaired by at most two rotations and $O(1)$ pointer writes.

6.19 A Complete AVL Tree in C++

Listing 6.9 is the whole structure: heights stored in the nodes, both rotations, the four-case rebalancer, insertion, deletion, a shape printer that shows every balance factor, and a self-check that verifies the invariant at every node. Its `main` replays all five insertion steps of §6.18 and the deletion example of §6.20, so every claim made by hand in this chapter is checked by a program.

Listing 6.9 A complete AVL tree. Compile with `g++ -std=c++17 -O2 -o avl avl.cpp` and run `./avl`.

```
// A complete AVL tree in C++: insertion with the four rotation cases,
// deletion with rebalancing all the way to the root, and a printer that
// shows every node's balance factor. main() replays the five insertion
// examples of Section 6.18 and the deletion example of Section 6.20.
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

class AVL {
public:
    AVL() : root_(nullptr), rotations_(0) {}
    ~AVL() { destroy(root_); }

    void insert(int key) { root_ = insertRec(root_, key); }
    void remove(int key) { root_ = removeRec(root_, key); }

    int  rotations() const { return rotations_; }
    void resetRotations()  { rotations_ = 0; }

    int height() const { return h(root_); }

    // key(balance factor){left subtree, right subtree}; "." is the empty tree.
    std::string shape() const { return shapeRec(root_); }

    bool isAVL() const { return checkRec(root_); }

private:
    struct Node {
        int key;
        int height;                 // height of a leaf is 0, of the empty tree -1
        Node* left;
        Node* right;
        Node(int k) : key(k), height(0), left(nullptr), right(nullptr) {}
    };
    Node* root_;
    int   rotations_;

    static int h(const Node* x) { return x ? x->height : -1; }
    static int bf(const Node* x) { return x ? h(x->right) - h(x->left) : 0; }
    static void refresh(Node* x) { x->height = 1 + std::max(h(x->left), h(x->right)); }

    static void destroy(Node* x) { if (!x) return; destroy(x->left); destroy(x->right);
delete x; }
```

Listing 6.9 (Continued) *A complete AVL tree. Compile with g++ -std=c++17 -O2 -o avl avl.cpp and run ./avl.*

```
/*      y                 x
 *     / \               / \
 *    x   P3    ==>    P1   y
 *   / \                   / \
 *  P1  P2               P2   P3
 */
Node* rotateRight(Node* y) {
    Node* x  = y->left;
    Node* P2 = x->right;
    x->right = y;               // y becomes the right child of x
    y->left  = P2;              // the middle subtree changes parent
    refresh(y);                 // y is now lower, so recompute it first
    refresh(x);
    ++rotations_;
    return x;                   // x is the new root of this subtree
}

/*    x                   y
 *   / \                 / \
 *  P1  y        ==>     x   P3
 *     / \              / \
 *    P2  P3           P1   P2
 */
Node* rotateLeft(Node* x) {
    Node* y  = x->right;
    Node* P2 = y->left;
    y->left  = x;
    x->right = P2;
    refresh(x);
    refresh(y);
    ++rotations_;
    return y;
}

Node* rebalance(Node* z) {
    refresh(z);
    int b = bf(z);
    if (b < -1) {                               // left side too tall
        if (bf(z->left) <= 0) return rotateRight(z);          // Left-Left
        z->left = rotateLeft(z->left);                        // Left-Right
        return rotateRight(z);
    }
    if (b > 1) {                                // right side too tall
        if (bf(z->right) >= 0) return rotateLeft(z);          // Right-Right
        z->right = rotateRight(z->right);                     // Right-Left
        return rotateLeft(z);
```

Listing 6.9 (Continued) *A complete AVL tree. Compile with g++ -std=c++17 -O2 -o avl avl.cpp and run ./avl.*

```
        }
        return z;                                   // already balanced
    }

    Node* insertRec(Node* x, int key) {
        if (!x) return new Node(key);
        if      (key < x->key) x->left  = insertRec(x->left,  key);
        else if (key > x->key) x->right = insertRec(x->right, key);
        else return x;                              // duplicate: no change
        return rebalance(x);
    }

    static Node* minNode(Node* x) { while (x->left) x = x->left; return x; }

    Node* removeRec(Node* x, int key) {
        if (!x) return nullptr;
        if      (key < x->key) x->left  = removeRec(x->left,  key);
        else if (key > x->key) x->right = removeRec(x->right, key);
        else {
            if (!x->left || !x->right) {            // no child, or exactly one
                Node* c = x->left ? x->left : x->right;
                delete x;
                if (!c) return nullptr;
                x = c;
            } else {
                Node* s = minNode(x->right);        // two children: inorder successor
                x->key   = s->key;
                x->right = removeRec(x->right, s->key);
            }
        }
        return rebalance(x);                        // rebalance on the way back up
    }

    static std::string shapeRec(const Node* x) {
        if (!x) return ".";
        if (!x->left && !x->right)
            return std::to_string(x->key) + "(" + std::to_string(bf(x)) + ")";
        return std::to_string(x->key) + "(" + std::to_string(bf(x)) + "){"
             + shapeRec(x->left) + "," + shapeRec(x->right) + "}";
    }
```

Listing 6.9 (Continued) *A complete AVL tree. Compile with g++ -std=c++17 -O2 -o avl avl.cpp and run ./avl.*

```
    static bool checkRec(const Node* x) {
        if (!x) return true;
        if (bf(x) < -1 || bf(x) > 1) return false;
        return checkRec(x->left) && checkRec(x->right);
    }
};

void step(const char* title, const std::vector<int>& build, int newKey) {
    AVL t;
    for (int k : build) t.insert(k);
    std::cout << title << "\n";
    std::cout << "  before      : " << t.shape() << "\n";
    t.resetRotations();
    t.insert(newKey);
    std::cout << "  insert " << newKey << "    : " << t.shape() << "\n";
    std::cout << "  rotations   : " << t.rotations()
              << ",  height " << t.height()
              << ",  still AVL: " << (t.isAVL() ? "yes" : "no") << "\n\n";
}

int main() {
    step("Step 1  no rebalancing needed", {13,10,15,5,11,16,4,6}, 14);
    step("Step 2  Left-Left case",        {13,10,15,5,11,16,4,8},  3);
    step("Step 3  Right-Right case",      {30,5,35,32,40},        45);
    step("Step 4  Left-Right case",       {13,10,15,5,11,16,4,6},  7);
    step("Step 5  Right-Left case",       {5,2,7,1,4,6,9,3,16},   15);

    AVL d;
    for (int k : {44,17,62,32,50,78,48,54,88}) d.insert(k);
    std::cout << "Deletion example\n";
    std::cout << "  before      : " << d.shape() << "\n";
    d.resetRotations();
    d.remove(32);
    std::cout << "  delete 32   : " << d.shape() << "\n";
    std::cout << "  rotations   : " << d.rotations()
              << ",  height " << d.height()
              << ",  still AVL: " << (d.isAVL() ? "yes" : "no") << "\n";
    return 0;
}
```

The program prints:

```
Step 1  no rebalancing needed
  before      : 13(-1){10(-1){5(0){4(0),6(0)},11(0)},15(1){.,16(0)}}
  insert 14    : 13(-1){10(-1){5(0){4(0),6(0)},11(0)},15(0){14(0),16(0)}}
  rotations   : 0,  height 3,  still AVL: yes

Step 2  Left-Left case
  before      : 13(-1){10(-1){5(0){4(0),8(0)},11(0)},15(1){.,16(0)}}
  insert 3    : 13(-1){5(0){4(-1){3(0),.},10(0){8(0),11(0)}},15(1){.,16(0)}}
  rotations   : 1,  height 3,  still AVL: yes

Step 3  Right-Right case
  before      : 30(1){5(0),35(0){32(0),40(0)}}
  insert 45    : 35(0){30(0){5(0),32(0)},40(1){.,45(0)}}
  rotations   : 1,  height 2,  still AVL: yes

Step 4  Left-Right case
  before      : 13(-1){10(-1){5(0){4(0),6(0)},11(0)},15(1){.,16(0)}}
  insert 7    : 13(-1){6(0){5(-1){4(0),.},10(0){7(0),11(0)}},15(1){.,16(0)}}
  rotations   : 2,  height 3,  still AVL: yes

Step 5  Right-Left case
  before      : 5(0){2(1){1(0),4(-1){3(0),.}},7(1){6(0),9(1){.,16(0)}}}
  insert 15    : 5(0){2(1){1(0),4(-1){3(0),.}},7(1){6(0),15(0){9(0),16(0)}}}
  rotations   : 2,  height 3,  still AVL: yes

Deletion example
  before      : 44(1){17(1){.,32(0)},62(0){50(0){48(0),54(0)},78(1){.,88(0)}}}
  delete 32   : 62(-1){44(1){17(0),50(0){48(0),54(0)}},78(1){.,88(0)}}
  rotations   : 1,  height 3,  still AVL: yes
```

Read the shape strings against the figures. Step 2's result, `13(-1){5(0){4(-1){3(0),.},10(0){8(0),11(0)}},15(1){.,16(0)}}`, is exactly the bottom-right tree of **Figure 6.15**: root 13, left child 5 with children 4 (holding 3) and 10 (holding 8 and 11), right child 15 with right child 16. Step 4 shows two rotations, as the LR case requires; step 5 likewise for RL; steps 2 and 3 show one each; step 1 shows none.

> **The only difference between Listing 6.4 and Listing 6.9** is a height field in the node, two rotation helpers, a nine-line `rebalance`, and one extra call at the end of `insertRec` and `removeRec`. That is the entire price of a worst-case guarantee. The search code is not touched at all — an AVL tree *is* a binary search tree, and searching it uses the same four lines of §6.7.

6.19.1 A reference implementation in Java

Figure 6.19 and **Figure 6.20** show a Java class, `AVLTree`, and it is worth looking at even in a C++ book, because it makes a design decision that our code repeats.

```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())`.

Two things in **Figure 6.19** deserve comment. First, the class comment says it all: an AVL tree *is* a tree, with one extra variable. That is the same relationship our **Listing 6.9** has to **Listing 6.4**. Second, the line `this.height = 1 + Math.max(left.getHeight(), right.getHeight())` is character for character the `refresh` function of **Listing 6.9**. Height is always one more than the taller child.

```java

    /**
     * Creates an empty tree
     */
    public AVLTree() {
        this.empty = true;
        this.value = 0;
        this.left = null;
        this.right = null;
        this.height = 0;
    }

    /**
     * Creates a tree with a single node
     */
    public AVLTree(int value) {
        this.empty = false;
        this.value = value;
        this.left = new AVLTree();
        this.right = new AVLTree();
        this.height = 1;
    }

    /**
     * Gets the height
     */
    private int getHeight() {
        return height;
    }

}
```

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 uses the *node-counting* height convention flagged back in §6.2: the empty tree has height 0 and a one-node tree has height 1. Our C++ uses −1 and 0. The balance factors, being differences, come out identical either way — which is exactly why the convention can be chosen freely, provided it is chosen once.

The Java implementation also represents the empty tree as a real object with an `empty` flag, rather than as a null pointer. That avoids null checks at the cost of allocating a sentinel for every missing child. Our C++ uses `nullptr` and checks for it in `h()`. Both designs are standard; the sentinel design is more common in languages without cheap null handling, and Weiss [1] uses a sentinel node of the same kind in his red-black trees.

6.20 AVL Deletion

Deletion in an AVL tree begins exactly as in a plain binary search tree — the three cases of §6.11 — and then rebalances. The procedure has one wrinkle that matters a great deal.

> **AVL DELETION.** Let s be the node being deleted.

1. Perform the standard binary-search-tree deletion of s.
2. Starting from s's old position, walk upward and find the first unbalanced node; call it z.
3. Let y be the child of z **with the greater height**, and let x be the child of y with the greater height. *(Note: this is a different rule from insertion, where y and x were determined by the path the new node took.)*
4. Rebalance the subtree rooted at z with the rotations its case requires — the same four cases, the same four figures: **Figure 6.10**, **Figure 6.11**, **Figure 6.12**, **Figure 6.13**.
5. **Keep going.** Unlike insertion, rebalancing z does not necessarily fix the whole tree; continue up to the root, rebalancing wherever necessary.

Steps 3 and 5 are the two differences, and both come from a single fact that we will now explain.

6.20.1 Why y and x are chosen differently

In insertion, the new node's path *is* the reason for the imbalance: the subtree that grew is the one the new node went into, so following the path finds the tall side automatically. In deletion, the imbalance is caused by a subtree that *shrank* — and the shrinking subtree is the short side, not the tall one. Rotations must move height from the tall side to the short side, so we must identify the tall side directly. Hence "the child with the greater height".

When the two children of y have equal height, either may be chosen as x; choose the one on the same side as y, so that the case is a single rotation, as the warning in §6.17.5 explains. The worked example of §6.20.3 meets exactly this tie.

6.20.2 Why insertion needs at most one rebalancing, and deletion may need lg$n$ of them

Return to the heights computed in §6.17. In all four insertion cases the subtree rooted at z had height $k+3$ after the insertion and height $k+2$ after the rotation — and $k+2$ was its height *before* the insertion. So from the point of view of z's parent, and of every ancestor above it, **nothing changed**: the subtree is exactly as tall as it was. No ancestor's balance factor can have moved, so no ancestor can be unbalanced. One rebalancing operation is always enough.

Deletion is different, and the difference is a single level. When a deletion shortens a subtree from height $k+1$ to k and a rotation is then performed at z, the rebalanced subtree can end up **shorter** than it was before the deletion. That shortening is itself a change of height, visible to z's parent — which may now be unbalanced in turn. The repair therefore propagates. In the worst case it propagates all the way to the root, one rotation per level, and since there are $O(\lg n)$ levels, deletion may perform $O(\lg n)$ rotations.

To say it once more, plainly: rebalancing node z does not rebalance the whole AVL tree; after rebalancing z the ancestors of z most probably need rebalancing too, and that is why the traversal must continue up to the root.

> **Both operations are still $O(\lg n)$.** A rotation costs $O(1)$, and there are at most $h = O(\lg n)$ of them, so deletion is $O(\lg n)$ overall — the same asymptotic cost as insertion. The difference is in the constant, and in a detail that matters for concurrent data structures: an insertion touches a bounded number of nodes after the descent, while a deletion may touch the entire path.

Listing 6.10 makes the difference visible. It builds **Fibonacci trees** — the minimum-node AVL trees of §6.15, which are the thinnest legal trees and therefore the most fragile — and measures the rotations caused by one insertion and by one deletion.

Listing 6.10 One insertion never costs more than one rebalancing; one deletion can cost a rebalancing at every level. Compile with `g++ -std=c++17 -O2 -o cascade cascade.cpp`.

```
// Why AVL insertion needs at most one rebalancing but AVL deletion may need
// one at every level. The worst case is the Fibonacci tree: the thinnest AVL
// tree of a given height. Deleting its largest key cascades to the root.
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <vector>
#include <cmath>

struct Node {
    int key, height;
    Node *left, *right;
    Node(int k) : key(k), height(0), left(nullptr), right(nullptr) {}
};

static int  h(Node* x)  { return x ? x->height : -1; }
static int  bf(Node* x) { return x ? h(x->right) - h(x->left) : 0; }
static void refresh(Node* x) { x->height = 1 + std::max(h(x->left), h(x->right)); }

int elementaryRotations = 0;   // single rotations performed
int rebalanceEvents     = 0;   // nodes at which a rotation case fired

Node* rotateRight(Node* y) {
    Node* x = y->left; Node* P2 = x->right;
    x->right = y; y->left = P2; refresh(y); refresh(x);
    ++elementaryRotations; return x;
}
Node* rotateLeft(Node* x) {
    Node* y = x->right; Node* P2 = y->left;
    y->left = x; x->right = P2; refresh(x); refresh(y);
    ++elementaryRotations; return y;
}

Node* rebalance(Node* z) {
    refresh(z);
    if (bf(z) < -1) {
        ++rebalanceEvents;
        if (bf(z->left) <= 0) return rotateRight(z);
        z->left = rotateLeft(z->left);
        return rotateRight(z);
    }
    if (bf(z) > 1) {
        ++rebalanceEvents;
        if (bf(z->right) >= 0) return rotateLeft(z);
        z->right = rotateRight(z->right);
        return rotateLeft(z);
```

Listing 6.10 (Continued) *One insertion never costs more than one rebalancing; one deletion can cost a rebalancing at every level. Compile with g++ -std=c++17 -O2 -o cascade cascade.cpp.*

```
    }
    return z;
}

Node* insertRec(Node* x, int key) {
    if (!x) return new Node(key);
    if      (key < x->key) x->left  = insertRec(x->left,  key);
    else if (key > x->key) x->right = insertRec(x->right, key);
    else return x;
    return rebalance(x);
}

Node* minNode(Node* x) { while (x->left) x = x->left; return x; }
Node* maxNode(Node* x) { while (x->right) x = x->right; return x; }

Node* removeRec(Node* x, int key) {
    if (!x) return nullptr;
    if      (key < x->key) x->left  = removeRec(x->left,  key);
    else if (key > x->key) x->right = removeRec(x->right, key);
    else {
        if (!x->left || !x->right) {
            Node* c = x->left ? x->left : x->right;
            delete x;
            if (!c) return nullptr;
            x = c;
        } else {
            Node* s = minNode(x->right);
            x->key = s->key;
            x->right = removeRec(x->right, s->key);
        }
    }
    return rebalance(x);
}

// The Fibonacci tree of height hh, keyed 1,2,3,... in inorder.
// Its left subtree has height hh-1 and its right subtree height hh-2:
// that is exactly the recurrence N(h) = N(h-1) + N(h-2) + 1.
Node* fibTree(int hh, int& counter) {
    if (hh < 0) return nullptr;
    if (hh == 0) { Node* n = new Node(++counter); refresh(n); return n; }
    Node* L = fibTree(hh - 1, counter);
    Node* r = new Node(++counter);
    Node* R = fibTree(hh - 2, counter);
    r->left = L; r->right = R; refresh(r);
    return r;
}
```

Listing 6.10 (Continued) *One insertion never costs more than one rebalancing; one deletion can cost a rebalancing at every level. Compile with g++ -std=c++17 -O2 -o cascade cascade.cpp.*

```
int size(Node* x) { return x ? 1 + size(x->left) + size(x->right) : 0; }
void destroy(Node* x) { if (!x) return; destroy(x->left); destroy(x->right); delete x; }

int main() {
    std::cout << "Insertion: rotations needed to add one key to a Fibonacci tree\n";
    std::cout << "  height   nodes   rebalance events   single rotations\n";
    for (int hh = 4; hh <= 16; hh += 3) {
        int c = 0;
        Node* t = fibTree(hh, c);
        int n = size(t);
        rebalanceEvents = elementaryRotations = 0;
        t = insertRec(t, 0);                      // a new smallest key
        std::cout << std::setw(8) << hh << std::setw(8) << n
                  << std::setw(19) << rebalanceEvents
                  << std::setw(19) << elementaryRotations << "\n";
        destroy(t);
    }

    std::cout << "\nDeletion: rotations needed to remove the largest key of a Fibonacci
tree\n";
    std::cout << "  height   nodes   rebalance events   single rotations   lg(n)\n";
    for (int hh = 4; hh <= 16; hh += 3) {
        int c = 0;
        Node* t = fibTree(hh, c);
        int n = size(t);
        int largest = maxNode(t)->key;
        rebalanceEvents = elementaryRotations = 0;
        t = removeRec(t, largest);
        std::cout << std::setw(8) << hh << std::setw(8) << n
                  << std::setw(19) << rebalanceEvents
                  << std::setw(19) << elementaryRotations
                  << std::setw(9) << std::fixed << std::setprecision(1)
                  << std::log2((double)n) << "\n";
        destroy(t);
    }
    return 0;
}
```

The program prints:

```
Insertion: rotations needed to add one key to a Fibonacci tree
  height   nodes   rebalance events   single rotations
       4      12                  1                  1
       7      54                  1                  1
      10     232                  1                  1
      13     986                  1                  1
      16    4180                  1                  1

Deletion: rotations needed to remove the largest key of a Fibonacci tree
  height   nodes   rebalance events   single rotations   lg(n)
       4      12                  2                  2      3.6
       7      54                  3                  3      5.8
      10     232                  5                  5      7.9
      13     986                  6                  6      9.9
      16    4180                  8                  8     12.0
```

The insertion column is a constant 1 no matter how tall the tree grows — exactly what the height argument of §6.20.2 predicted. The deletion column grows with the height: 2,3,5,6,8 rotations for heights 4,7,10,13,16, which is about $h/2$, and therefore $\Theta(\lg n)$.

Notice also that the node counts in the first column of each table — 12,54,232,986,4180 — are the values $N(4), N(7), N(10), N(13), N(16)$ from the recurrence of §6.15. Rows $h = 4$, 7 and 10 can

be checked directly against **Table 6.4**: 12, 54 and 232. The program built the trees from the recurrence and the recurrence produced the sizes the table predicts.

6.20.3 A deletion example, step by step

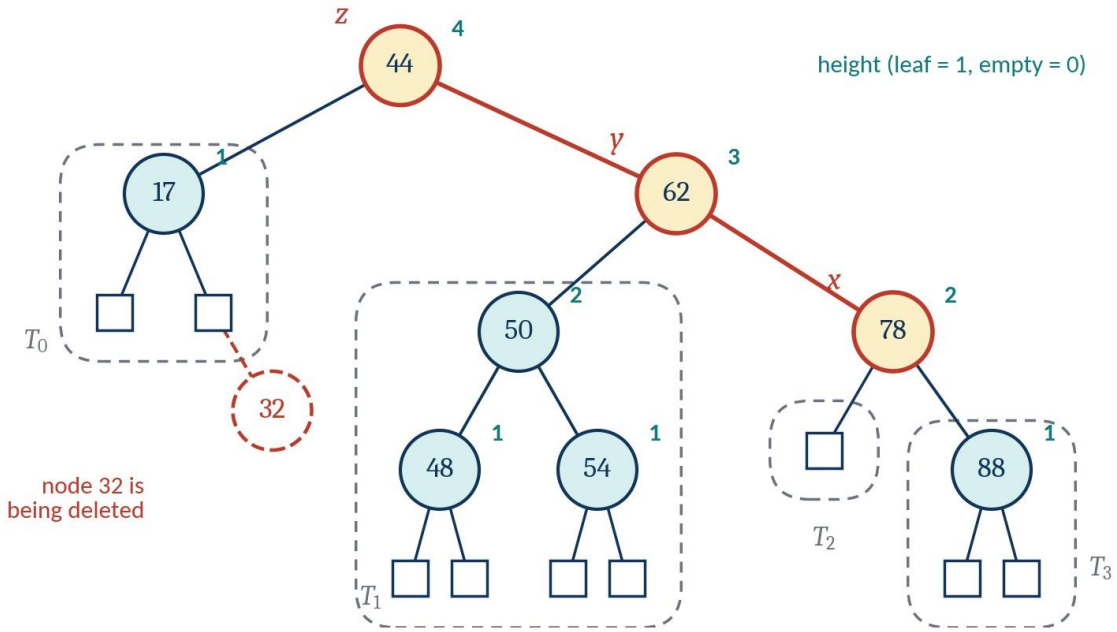


Figure 6.21 Deleting from an AVL tree: the node 32 is about to be deleted. Heights are written beside the nodes, and the first unbalanced ancestor z, its taller child y and y's taller child x are marked in red. The empty squares are null subtrees, and T_0 to T_3 name the four subtrees the rotation will move.

The tree of **Figure 6.21** is: root 44; left child 17, which has a right child 32; right child 62, whose left child is 50 (with children 48 and 54) and whose right child is 78 (with a right child 88).

Note the height convention in the figure: a leaf is marked 1 and a null subtree 0. That is the node-counting convention of §6.2, one greater than ours everywhere. We will quote the figure's numbers as the figure gives them and our program's numbers in our own convention; the balance factors agree.

Step 1: delete 32 **by the ordinary BST rule.** Node 32 is a leaf, so this is case 1 of §6.11: detach it from its parent 17 and free it. Node 17 becomes a leaf, and in the figure's convention its height drops from 2 to 1.

Step 2: walk up and find the first unbalanced node. At node 17: both subtrees empty, balanced. At node 44: its left subtree is now just 17, of height 1 in the figure's numbering, while its right subtree rooted at 62 has height 3. The difference is 2, so 44 is unbalanced: after deleting 32 we go up and find that the first unbalanced node is 44. So $z = 44$.

Step 3: choose y and x by height. The children of $z = 44$ are 17 (height 1) and 62 (height 3). The taller is 62, so $y = 62$. The children of $y = 62$ are 50 (height 2) and 78 (height 2) — *a tie*. Either could serve as x, because both have the same height; we choose 78, the child on the same side as y, which makes the case a single rotation. So $x = 78$.

Step 4: identify the case. $y = 62$ is the **right** child of $z = 44$, and $x = 78$ is the **right** child of y. Right-Right, so a single left rotation will be done.

Step 5: perform one left rotation about $z = 44$, exactly as **Figure 6.12** shows.

1. $y \leftarrow$ 44->right $= 62$.

2. $P_2 \leftarrow$ 62->left $= 50$ — this is the subtree labeled T_1 in the figures.
3. 62->left $\leftarrow$ 44.
4. 44->right $\leftarrow$ 50.
5. Refresh 44, then 62.

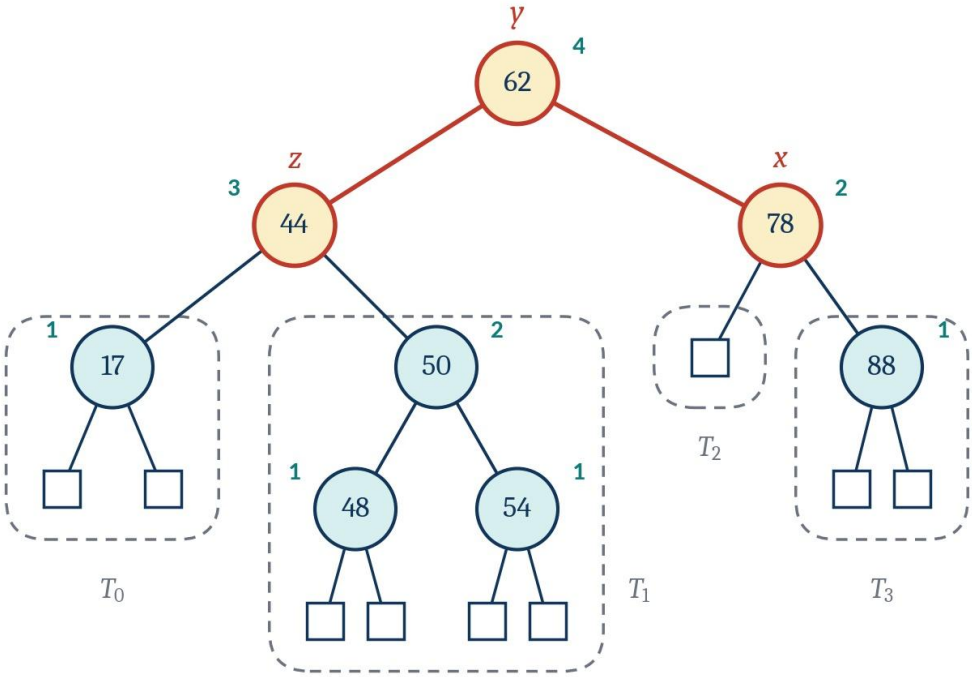


Figure 6.22 The tree after the left rotation. Node 62 is the new root; 44 has dropped to become its left child and has taken the subtree T_1 (rooted at 50) as its right child; 78 is unchanged as the right child of 62.

Figure 6.22 is the result. Node 62 is now the root with height 4 in the figure's convention; 44 is its left child with height 3, holding 17 (T_0) on the left and 50 (T_1) on the right; 78 is its right child with height 2, holding the null subtree T_2 on the left and 88 (T_3) on the right.

Check the invariant at every node, in the figure's convention:

- 17: two null children, heights 0 and 0. Balanced.
- 48, 54, 88: leaves. Balanced.
- 50: children 48 and 54, heights 1 and 1. Balanced.
- 78: children T_2 (height 0) and 88 (height 1). Difference 1. Balanced.
- 44: children 17 (height 1) and 50 (height 2). Difference 1. Balanced.
- 62: children 44 (height 3) and 78 (height 2). Difference 1. Balanced.

Every node is legal, so one rotation sufficed here. The last block of output from **Listing 6.9** confirms the whole thing:

```
Deletion example
  before      : 44(1){17(1){.,32(0)},62(0){50(0){48(0),54(0)},78(1){.,88(0)}}}
  delete 32   : 62(-1){44(1){17(0),50(0){48(0),54(0)}},78(1){.,88(0)}}
  rotations   : 1,  height 3,  still AVL: yes
```

Read the second shape string against **Figure 6.22**: root 62 with balance factor -1; left child 44 with balance factor $+1$, holding the leaf 17 and the subtree 50{48,54}; right child 78 with balance factor $+1$, holding nothing on the left and 88 on the right. It is the figure, written on one line.

6.21 Choosing: AVL Tree, Red-Black Tree, or Hash Table

An AVL tree is not always the right answer. Two other structures compete for the same job, and a professional chooses between them deliberately.

6.21.1 Red-black trees

A **red-black tree** is another self-balancing binary search tree. Instead of storing a height at each node it stores one bit — a color, red or black — and maintains five rules [2], of which the important ones are: the root is black; a red node never has a red child; and every path from a node down to a missing child passes through the same number of black nodes.

That last rule does the balancing. If every root-to-leaf path has the same number of black nodes, and no two reds are adjacent, then the longest path can be at most twice the shortest: alternate red and black on the long path, all black on the short one. Hence

$$h \leq 2\lg(n+1) \tag{6.11}$$

for a red-black tree (Equation (6.11)), against $h \leq 1.4405\lg(n+2) - 1.3277$ for an AVL tree (Equation (6.9)). **The AVL tree is strictly the shorter of the two.**

Why, then, is `std::map` in the C++ standard library a red-black tree and not an AVL tree? Because the bound on *height* is not the only bound that matters. Red-black trees are looser, and looseness means they need to fix themselves less often:

- a red-black **insertion** performs at most 2 rotations, then only recolorings;
- a red-black **deletion** performs at most 3 rotations, then only recolorings;
- an AVL **insertion** performs at most 2 elementary rotations;
- an AVL **deletion** may perform $\Theta(\lg n)$ rotations, as **Listing 6.10** demonstrated.

So: AVL trees give faster lookups, because the tree is shorter; red-black trees give faster modification, because the repair is bounded. The rule of thumb every practitioner uses is therefore:

> **Read-heavy workload — many searches, few changes — favors the AVL tree. Write-heavy workload — constant insertion and deletion — favors the red-black tree.** A database index that is rebuilt nightly and queried all day is an AVL-shaped problem; an in-memory map that is churned continuously is a red-black-shaped problem.

6.21.2 Hash tables

A **hash table** answers a completely different question. It computes a function of the key, uses the result as an array index, and jumps straight there. Its expected cost for insert, lookup and delete is $O(1)$ — not $O(\lg n)$, but $O(1)$, independent of n.

That is an enormous advantage and you should take it whenever you can. But it is bought by throwing away order. A hash function deliberately scatters keys, so in a hash table:

- there is no meaningful "next key after k" — the successor operation is gone;
- there is no minimum and no maximum without scanning every bucket;
- you cannot list the keys in sorted order without extracting and sorting them, at $\Theta(n\lg n)$;
- you cannot answer a **range query** — "every key between a and b" — without examining the entire table;

- and the $O(1)$ is an *expectation*, not a guarantee: a bad hash function or an adversarial input can degrade a hash table to $\Theta(n)$ per operation, and rehashing causes occasional expensive insertions.

A balanced tree keeps all of those abilities, at $O(\lg n)$ per operation, guaranteed.

Listing 6.11 measures the trade on 200 000 keys: an AVL tree, a plain BST, `std::map` (a red-black tree) and `std::unordered_map` (a hash table).

Listing 6.11 Measuring an AVL tree against a plain BST, a red-black tree and a hash table. Compile with `g++ -std=c++17 -O2 -o compare compare.cpp`.

```
// A measured comparison: plain BST, AVL tree, std::map (a red-black tree),
// and std::unordered_map (a hash table). The last test is a range query,
// which the hash table cannot answer without looking at every key.
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <chrono>
#include <map>
#include <unordered_map>
#include <random>
#include <vector>

struct Node {
    int key, height;
    Node *left, *right;
    Node(int k) : key(k), height(0), left(nullptr), right(nullptr) {}
};
static int  h(Node* x)  { return x ? x->height : -1; }
static int  bf(Node* x) { return x ? h(x->right) - h(x->left) : 0; }
static void refresh(Node* x) { x->height = 1 + std::max(h(x->left), h(x->right)); }
Node* rotR(Node* y) { Node* x=y->left;  Node* t=x->right; x->right=y; y->left =t;
refresh(y); refresh(x); return x; }
Node* rotL(Node* x) { Node* y=x->right; Node* t=y->left;  y->left =x; x->right=t;
refresh(x); refresh(y); return y; }
Node* balance(Node* z) {
    refresh(z);
    if (bf(z) < -1) { if (bf(z->left)  > 0) z->left  = rotL(z->left);  return rotR(z); }
    if (bf(z) >  1) { if (bf(z->right) < 0) z->right = rotR(z->right); return rotL(z); }
    return z;
}
Node* avlInsert(Node* x, int k) {
    if (!x) return new Node(k);
    if      (k < x->key) x->left  = avlInsert(x->left,  k);
    else if (k > x->key) x->right = avlInsert(x->right, k);
    else return x;
    return balance(x);
}
Node* bstInsert(Node* x, int k) {
    if (!x) return new Node(k);
    if      (k < x->key) x->left  = bstInsert(x->left,  k);
    else if (k > x->key) x->right = bstInsert(x->right, k);
    return x;
}
bool findIter(Node* x, int k) { while (x && x->key != k) x = (k < x->key) ? x->left : x-
>right; return x != nullptr; }
int  heightOf(Node* x) { return x ? 1 + std::max(heightOf(x->left), heightOf(x->right)) : -
1; }
void destroy(Node* x) { if (!x) return; destroy(x->left); destroy(x->right); delete x; }
// Sum the keys in [lo, hi] without visiting subtrees that cannot contain them.
long long rangeSum(Node* x, int lo, int hi) {
```

Listing 6.11 (Continued) *Measuring an AVL tree against a plain BST, a red-black tree and a hash table. Compile with g++ -std=c++17 -O2 -o compare compare.cpp.*

```
    if (!x) return 0;
    if (x->key < lo)  return rangeSum(x->right, lo, hi);
    if (x->key > hi)  return rangeSum(x->left,  lo, hi);
    return (long long)x->key + rangeSum(x->left, lo, hi) + rangeSum(x->right, lo, hi);
}

using Clock = std::chrono::steady_clock;
static double ms(Clock::time_point a, Clock::time_point b) {
    return std::chrono::duration<double, std::milli>(b - a).count();
}

int main() {
    const int n = 200000;
    std::vector<int> ascending(n);
    for (int i = 0; i < n; ++i) ascending[i] = i;
    std::vector<int> shuffled(ascending);          // a copy, then shuffled below
    std::mt19937 rng(2026);
    std::shuffle(shuffled.begin(), shuffled.end(), rng);

    std::cout << std::fixed << std::setprecision(2);

    // 1. Ascending keys: the plain BST degenerates, the AVL tree does not.
    Node* a = nullptr;
    auto t0 = Clock::now();
    for (int k : ascending) a = avlInsert(a, k);
    auto t1 = Clock::now();
    std::cout << "AVL, ascending keys   : height " << std::setw(6) << heightOf(a)
              << "   insert " << std::setw(8) << ms(t0, t1) << " ms\n";

    Node* b = nullptr;
    t0 = Clock::now();
    for (int k : shuffled) b = bstInsert(b, k);
    t1 = Clock::now();
    std::cout << "BST, shuffled keys    : height " << std::setw(6) << heightOf(b)
              << "   insert " << std::setw(8) << ms(t0, t1) << " ms\n";

    std::map<int,int> rb;
    t0 = Clock::now();
    for (int k : ascending) rb[k] = k;
    t1 = Clock::now();
    std::cout << "std::map, ascending   : size   " << std::setw(6) << rb.size()
              << "   insert " << std::setw(8) << ms(t0, t1) << " ms\n";
```

Listing 6.11 (Continued) *Measuring an AVL tree against a plain BST, a red-black tree and a hash table. Compile with g++ -std=c++17 -O2 -o compare compare.cpp.*

```
    std::unordered_map<int,int> hm;
    t0 = Clock::now();
    for (int k : ascending) hm[k] = k;
    t1 = Clock::now();
    std::cout << "unordered_map, asc.   : size   " << std::setw(6) << hm.size()
              << "   insert " << std::setw(8) << ms(t0, t1) << " ms\n\n";

    // 2. Lookups of keys that are all present.
    long long found = 0;
    t0 = Clock::now(); for (int k : shuffled) found += findIter(a, k);  t1 = Clock::now();
    std::cout << "AVL lookup            : " << std::setw(8) << ms(t0, t1) << " ms  (" <<
found << " hits)\n";
    found = 0;
    t0 = Clock::now(); for (int k : shuffled) found += (rb.find(k) != rb.end()); t1 =
Clock::now();
    std::cout << "std::map lookup       : " << std::setw(8) << ms(t0, t1) << " ms  (" <<
found << " hits)\n";
    found = 0;
    t0 = Clock::now(); for (int k : shuffled) found += (hm.find(k) != hm.end()); t1 =
Clock::now();
    std::cout << "unordered_map lookup  : " << std::setw(8) << ms(t0, t1) << " ms  (" <<
found << " hits)\n\n";

    // 3. A range query: sum every key between 90000 and 90999.
    long long s1 = 0, s2 = 0, s3 = 0;
    t0 = Clock::now(); s1 = rangeSum(a, 90000, 90999); t1 = Clock::now();
    std::cout << "AVL range sum         : " << std::setw(8) << ms(t0, t1) << " ms  (sum "
<< s1 << ")\n";
    t0 = Clock::now();
    for (auto it = rb.lower_bound(90000); it != rb.end() && it->first <= 90999; ++it) s2 +=
it->first;
    t1 = Clock::now();
    std::cout << "std::map range sum    : " << std::setw(8) << ms(t0, t1) << " ms  (sum "
<< s2 << ")\n";
    t0 = Clock::now();
    for (const auto& kv : hm) if (kv.first >= 90000 && kv.first <= 90999) s3 += kv.first;
    t1 = Clock::now();
    std::cout << "hash table range sum  : " << std::setw(8) << ms(t0, t1) << " ms  (sum "
<< s3 << ")\n";

    destroy(a); destroy(b);
    return 0;
}
```

On the machine used to prepare this chapter the program printed:

```
AVL, ascending keys   : height     17   insert    38.18 ms
BST, shuffled keys    : height     41   insert   145.14 ms
std::map, ascending   : size   200000   insert    40.73 ms
unordered_map, asc.   : size   200000   insert    10.57 ms

AVL lookup            :    36.59 ms  (200000 hits)
std::map lookup       :   107.89 ms  (200000 hits)
unordered_map lookup  :     2.65 ms  (200000 hits)

AVL range sum         :     0.01 ms  (sum 90499500)
std::map range sum    :     0.01 ms  (sum 90499500)
hash table range sum  :     0.51 ms  (sum 90499500)
```

Timings vary between machines, but the shape of the result does not. Read it in three parts.

Heights. Fed 200 000 keys in *ascending* order — the worst case for a plain BST — the AVL tree reached height 17. Our bound Equation (6.9) predicts at most $1.4405\lg(200002) - 1.3277 \approx$

23.1, so 17 is comfortably inside it, and lg200000 ≈ 17.6 says 17 is essentially optimal. The plain BST, fed *shuffled* keys — its best realistic case — still reached height 41, more than twice as deep. A plain BST fed ascending keys would have reached height 199 999; that run is omitted only because it takes minutes rather than milliseconds.

Lookups. The hash table wins by an order of magnitude: 2.65 ms against 36.59 ms. When all you do is look up single keys, a hash table is the right structure and no tree will beat it.

Range queries. The AVL tree and `std::map` answer "sum every key from 90 000 to 90 999" in 0.01 ms, because both can descend to the start of the range and then walk only the 1000 keys inside it. The hash table needs 0.51 ms — about 51 times longer — because it has to look at all 200 000 entries to discover which 1000 qualify. Widen the table to twenty million keys and that gap becomes a factor of thousands.

Table 6.6 is the decision table to keep.

Table 6.6 Choosing an ordered dictionary. k is the number of keys the range query actually returns.

Requirement	Plain BST	AVL tree	Red-black tree	Hash table
Search, worst case	$O(n)$	$O(\lg n)$	$O(\lg n)$	$O(n)$
Search, typical	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	$O(1)$
Insert / delete, worst case	$O(n)$	$O(\lg n)$	$O(\lg n)$	$O(n)$, amortized $O(1)$
Rotations per insert	none	≤ 2	≤ 2	—
Rotations per delete	none	$O(\lg n)$	≤ 3	—
Guaranteed height	none	$\approx 1.44 \lg n$	$\leq 2 \lg n$	—
Keys in sorted order	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \lg n)$
Minimum / maximum	$O(h)$	$O(\lg n)$	$O(\lg n)$	$\Theta(n)$
Successor / predecessor	$O(h)$	$O(\lg n)$	$O(\lg n)$	not supported
Range query $[a, b]$	$O(h + k)$	$O(\lg n + k)$	$O(\lg n + k)$	$\Theta(n)$
Extra space per node	2 pointers	2 pointers + a height	2 pointers + 1 bit	bucket array + chains
Choose it when	never, in production	reads dominate writes	writes are frequent	only point lookups are needed

6.22 Profitina in Practice

Profitina is an **AI-visibility (GEO) platform**: it scans a brand's website, scores how ready that site is to be *cited by an AI answer engine*, explains each signal it measured, and then helps fix what it found. It runs in production at `app.profitina.com`, on Python and PostgreSQL, with local model inference through Ollama and Qwen2.5. Everything in this chapter appears inside it, in four distinct places.

The crawl frontier is an ordered structure, not a queue. A scan of a customer's site discovers URLs faster than it can fetch them. Those pending URLs are not processed first-come-first-served: they are processed in order of a priority that combines depth, estimated importance and politeness delay. A structure that must repeatedly yield "the most urgent item" and also accept new items at any moment is precisely the ordered-dictionary problem of this chapter. Where the scheduler additionally needs *ranges* — "every URL whose politeness timer expires in the next

thirty seconds" — a balanced search tree earns its keep, because a heap cannot answer that and a hash table cannot either.

The re-scan cadence is a successor query. Every brand on a paid tier is re-scanned automatically at a per-tier cadence, and the scheduler must repeatedly ask: *which scan is due next, and when?* That is TREE-MINIMUM on a tree keyed by due-time. When a scan finishes and is rescheduled, that is a delete followed by an insert. When an operator asks "what is due between 02:00 and 04:00 tonight?", that is a range query — $O(\lg n + k)$ in a balanced tree and $\Theta(n)$ in a hash table. This is exactly the shape of **Listing 6.12** below.

The weekly score trend is an inorder walk. The historical analytics dashboard shows a brand's visibility score over time and exports it as CSV. The export is an inorder traversal of a time-keyed index: $\Theta(n)$, in sorted order, with no sorting step. The proof in §6.6.1 is the reason the CSV comes out in date order without anybody calling a sort.

URL canonicalization is a hash-table problem, deliberately. Deduplicating `https://example.co.id/page`, `http://example.co.id/page/` and `https://www.example.co.id/page?utm_source=x` into one canonical page needs only *equality* — never "the next URL after this one". That is the row of **Table 6.6** where a hash table wins outright, and the production system uses one. Knowing when *not* to reach for a tree is as much a part of this chapter as knowing when to.

> **Two rules we do not bend.** First, the platform advises; the customer decides. A visibility score is a measurement with an explanation attached, never a promise about how an answer engine will behave. Second, anything touching personal data follows UU PDP 27/2022 — explicit consent, data export, and complete account purge. A complete purge means every index holding a reference to that account must support deletion, which is one practical reason the data structures in the system are ones that can delete cheaply, not append-only ones.

6.22.1 The worked case study: Profitina Laundry

The runnable case study in this book is always **Profitina Laundry**, a small neighborhood laundry in Surabaya — not the Profitina platform itself.

The laundry takes bags of washing across the counter all day. Each bag gets a ticket, a customer name, a weight in kilograms, and a **promised pickup time**. The staff behind the counter need four things, constantly:

1. file a newly accepted bag under its promised minute;
2. answer "what is due next?" when the shift is quiet and they want to get ahead;
3. answer "what is promised between 10:00 and 14:30?" when planning the machine loads for a block of the afternoon;
4. remove a bag from the board when the customer collects it.

Question 2 is a **successor** query. Question 3 is a **range** query. Questions 1 and 4 are insertion and deletion. A hash table answers none of the first three. A sorted array answers 2 and 3 but is $\Theta(n)$ for 1 and 4. An AVL tree keyed by promised minute answers all four in $O(\lg n)$, guaranteed, even though tickets arrive all day in increasing time order — the very pattern that destroys a plain binary search tree.

Listing 6.12 is the pickup board. Note two details. The key is the pair (minute, ticket), so two bags promised for the same minute are still distinct keys and the tree never has to deal with duplicates.

And `nextDueAfter` performs the successor search with no parent pointers at all, by carrying the best candidate down from the root — the second technique mentioned in §6.9.

Listing 6.12 The Profitina Laundry pickup board: an AVL tree keyed by promised pickup time, supporting a successor query, a range query and deletion. Compile with `g++ -std=c++17 -O2 -o laundry laundry.cpp`.

```
// Profitina Laundry: the pickup board.
// Every order is filed under the minute it is promised for. The counter staff
// need four things: file a new order, find the next order due after "now",
// list everything due inside a time window, and remove an order once collected.
// An AVL tree keyed by promised minute answers all four in O(lg n) or better.
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <cstdio>
#include <string>
#include <vector>

struct Order {
    int         minute;      // minutes after midnight the laundry is promised
    std::string ticket;      // the ticket number written on the bag
    std::string customer;
    int         kilos;
};

struct Node {
    Order data;
    int   height;
    Node* left;
    Node* right;
    Node(const Order& o) : data(o), height(0), left(nullptr), right(nullptr) {}
};

static int  h(Node* x)  { return x ? x->height : -1; }
static int  bf(Node* x) { return x ? h(x->right) - h(x->left) : 0; }
static void refresh(Node* x) { x->height = 1 + std::max(h(x->left), h(x->right)); }

Node* rotateRight(Node* y) { Node* x=y->left;  Node* t=x->right; x->right=y; y->left =t;
refresh(y); refresh(x); return x; }
Node* rotateLeft (Node* x) { Node* y=x->right; Node* t=y->left;  y->left =x; x->right=t;
refresh(x); refresh(y); return y; }

Node* rebalance(Node* z) {
    refresh(z);
    if (bf(z) < -1) { if (bf(z->left)  > 0) z->left  = rotateLeft(z->left);   return
rotateRight(z); }
    if (bf(z) >  1) { if (bf(z->right) < 0) z->right = rotateRight(z->right); return
rotateLeft(z);  }
    return z;
}
```

Listing 6.12 (Continued) *The Profitina Laundry pickup board: an AVL tree keyed by promised pickup time, supporting a successor query, a range query and deletion. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
// Two orders promised for the same minute are separated by their ticket
// number, so every key in the tree is distinct.
static int compareKey(int m1, const std::string& t1, int m2, const std::string& t2) {
    if (m1 != m2) return m1 < m2 ? -1 : 1;
    if (t1 == t2) return 0;
    return t1 < t2 ? -1 : 1;
}

Node* fileOrder(Node* x, const Order& o) {
    if (!x) return new Node(o);
    int c = compareKey(o.minute, o.ticket, x->data.minute, x->data.ticket);
    if (c < 0) x->left  = fileOrder(x->left,  o);
    else       x->right = fileOrder(x->right, o);
    return rebalance(x);
}

// The first order promised strictly after "now": a successor search from the root.
const Order* nextDueAfter(Node* x, int now) {
    const Order* best = nullptr;
    while (x) {
        if (x->data.minute > now) { best = &x->data; x = x->left; }  // candidate; try
earlier
        else                      { x = x->right; }                  // too early; go later
    }
    return best;
}

void listWindow(Node* x, int lo, int hi, std::vector<const Order*>& out) {
    if (!x) return;
    if (x->data.minute >= lo) listWindow(x->left, lo, hi, out);   // skip the left subtree
if it is all too early
    if (x->data.minute >= lo && x->data.minute <= hi) out.push_back(&x->data);
    if (x->data.minute <= hi) listWindow(x->right, lo, hi, out);  // skip the right subtree
if it is all too late
}

Node* minNode(Node* x) { while (x->left) x = x->left; return x; }

Node* collectOrder(Node* x, int minute, const std::string& ticket, bool& done) {
    if (!x) return nullptr;
    int c = compareKey(minute, ticket, x->data.minute, x->data.ticket);
    if      (c < 0) x->left  = collectOrder(x->left,  minute, ticket, done);
    else if (c > 0) x->right = collectOrder(x->right, minute, ticket, done);
    else {
        done = true;
        if (!x->left || !x->right) {                 // no child, or exactly one
            Node* c2 = x->left ? x->left : x->right;
            delete x;
            if (!c2) return nullptr;
```

Listing 6.12 (Continued) *The Profitina Laundry pickup board: an AVL tree keyed by promised pickup time, supporting a successor query, a range query and deletion. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
            x = c2;
        } else {                                    // two children
            Node* s = minNode(x->right);            // the inorder successor
            Order moved = s->data;
            bool inner = false;
            x->right = collectOrder(x->right, moved.minute, moved.ticket, inner);
            x->data  = moved;
        }
    }
    return rebalance(x);
}

std::string clock24(int minute) {
    char buf[16];
    std::snprintf(buf, sizeof buf, "%02d:%02d", minute / 60, minute % 60);
    return buf;
}

void show(const Order& o) {
    std::cout << "  " << clock24(o.minute) << "  " << std::left << std::setw(7) << o.ticket
              << std::setw(10) << o.customer << std::right << std::setw(3) << o.kilos << "
kg\n";
}

int main() {
    Node* board = nullptr;
    std::vector<Order> intake = {
        {11*60+30, "PL-014", "Dewi",    4},
        { 9*60+15, "PL-002", "Bagus",   7},
        {16*60+00, "PL-031", "Sari",    3},
        {10*60+45, "PL-008", "Joko",    5},
        {13*60+20, "PL-019", "Rina",    9},
        { 8*60+30, "PL-001", "Agus",    2},
        {17*60+45, "PL-036", "Putu",    6},
        {14*60+10, "PL-023", "Mega",    8},
        {12*60+00, "PL-016", "Hendra", 11},
    };
    for (const Order& o : intake) board = fileOrder(board, o);
    std::cout << "orders on the board : " << intake.size()
              << ",  tree height : " << h(board) << "\n\n";

    int now = 12*60 + 30;
    std::cout << "It is " << clock24(now) << ". The next bag due is:\n";
    const Order* nxt = nextDueAfter(board, now);
    if (nxt) show(*nxt); else std::cout << "  nothing left today\n";
```

Listing 6.12 (Continued) *The Profitina Laundry pickup board: an AVL tree keyed by promised pickup time, supporting a successor query, a range query and deletion. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
    std::cout << "\nEverything promised between 10:00 and 14:30:\n";
    std::vector<const Order*> win;
    listWindow(board, 10*60, 14*60+30, win);
    for (const Order* o : win) show(*o);

    std::cout << "\nRina collects PL-019 at 13:20.\n";
    bool done = false;
    board = collectOrder(board, 13*60+20, "PL-019", done);
    std::cout << "removed: " << (done ? "yes" : "no")
              << ",  tree height now : " << h(board) << "\n";

    std::cout << "\nThe window again, after the collection:\n";
    win.clear();
    listWindow(board, 10*60, 14*60+30, win);
    for (const Order* o : win) show(*o);
    return 0;
}
```

The program prints:

```
orders on the board : 9,  tree height : 3

It is 12:30. The next bag due is:
  13:20  PL-019 Rina        9 kg

Everything promised between 10:00 and 14:30:
  10:45  PL-008 Joko        5 kg
  11:30  PL-014 Dewi        4 kg
  12:00  PL-016 Hendra     11 kg
  13:20  PL-019 Rina        9 kg
  14:10  PL-023 Mega        8 kg

Rina collects PL-019 at 13:20.
removed: yes,  tree height now : 3

The window again, after the collection:
  10:45  PL-008 Joko        5 kg
  11:30  PL-014 Dewi        4 kg
  12:00  PL-016 Hendra     11 kg
  14:10  PL-023 Mega        8 kg
```

Nine orders sit in a tree of height 3. Our bound allows at most $1.4405\lg 11 - 1.3277 \approx 3.6$, so 3 is the tallest legal value; and since a tree of height 2 holds at most 7 nodes, 3 is also the shortest possible. The tree is exactly as short as nine nodes allow. The window listing comes out in time order without any sorting step — it is an inorder walk restricted to a range. And the two subtree tests inside `listWindow` are what make the query $O(\lg n + k)$ rather than $\Theta(n)$: if a node's time is already earlier than the window, its entire left subtree is earlier still and is never visited.

6.23 Common Mistakes, and How to Avoid Them

1. Checking the BST property only against the immediate children. This is the single most common bug, and it is the trap introduced in §6.3. The test "left child smaller, right child larger" is necessary but nowhere near sufficient. The correct test carries a permitted open interval down the tree: everything in the left subtree of x must lie in $(\text{lo},\text{key}(x))$ and everything in the right subtree in $(\text{key}(x),\text{hi})$. **Listing 6.13** shows both tests on a tree that fools the first one.

Listing 6.13 The wrong and the right way to verify the binary-search-tree property. Compile with `g++ -std=c++17 -O2 -o isbst isbst.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 6.13 (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;
}
```

The program prints:

```
bad  tree: local test says BST, correct test says not a BST
good tree: local test says BST, correct test says BST
```

The first line is the whole lesson: a tree that passes the local test and fails the real one. The interval is passed as `long long` so that the sentinels `LLONG_MIN` and `LLONG_MAX` can lie strictly outside the range of any `int` key.

2. Refreshing heights in the wrong order after a rotation. In `rotateRight(y)`, the node y ends up *below* x, so `refresh(y)` must run before `refresh(x)`. Reverse the two lines and x's height will be computed from y's stale value. The tree stays a correct search tree, the invariant check may even pass, and yet the heights drift — and the drift makes the rebalancer fire at the wrong places. This bug does not crash; it merely makes the tree slowly worse. Add an assertion that recomputes heights from scratch and compares.

3. Mixing the two height conventions. If `height(nullptr)` returns 0 in one function and -1 in another, every balance factor near a leaf is wrong by one and the tree will either rotate when it should not or fail to rotate when it should. Define one helper `h()` and never read `x->height` directly on a pointer that might be null.

4. Forgetting that deletion must keep going. Insertion may stop after one rebalancing, but deletion may not — §6.20.2 proved it and **Listing 6.10** measured it. Code that rebalances only the first unbalanced ancestor and then returns will produce trees that violate the invariant. The recursive formulation of **Listing 6.9** gets this right for free, because `rebalance` is called on *every* node on the way back up.

5. Confusing the BST property with the heap property. Covered in §6.4 and in **Table 6.1**. If you catch yourself writing "the largest key is at the root" about a search tree, stop: the largest key of a search tree is at the far right, and the root is somewhere in the middle.

6. Assuming a BST is balanced because it usually is. A plain binary search tree is $\Theta(\lg n)$ on random input and $\Theta(n)$ on sorted input, and real data is far more often sorted than random. **Listing 6.5** measured a factor of five hundred. If the code must never be slow, use a balanced tree; if it merely must usually be fast, you are gambling with data you do not control.

7. Copying a key but not its satellite data. In case 3 of deletion, `x->key = s->key` moves the key up. If the node also holds a customer record, that record must move too. Forgetting this produces a tree whose keys are right and whose payloads belong to somebody else — a bug that no invariant check will ever find.

8. Deleting a node while it is still reachable. In the one-child case, promote the child *before* calling `delete`, and never dereference the pointer afterward. The idiom in **Listing 6.4** and **Listing 6.9** — save the child, delete the node, return the child — is safe; reordering it is not.

9. Treating duplicate keys as an afterthought. This chapter forbids duplicates, and both implementations silently ignore a repeated key. If your application needs duplicates, decide deliberately: store a count in the node, or extend the key with a tie-breaker as **Listing 6.12** does with the ticket number. What you must not do is let equal keys drift to both sides, because then search can no longer tell which side to take.

10. Using a balanced tree where a hash table would do. The last row of **Table 6.6** works in both directions. If your operations are only "insert", "look up" and "erase" a single key, a hash table is simpler and about ten times faster, as **Listing 6.11** measured. Trees earn their extra cost only when order matters.

6.24 Chapter Summary

A **binary search tree** is a binary tree in which every key in a node's left subtree is smaller than the node, and every key in its right subtree is larger — *every* key, not merely the children. That one rule, stated precisely in §6.3, makes each comparison discard an entire side of the collection.

The **heap property** is a different rule with a different purpose, and §6.4 compared them line by line. A heap finds the extreme fast; a search tree finds anything fast.

There are three **traversals**. The inorder walk prints the keys in sorted order, and §6.6.1 proved it by induction on the number of nodes. The preorder walk rebuilds a tree's exact shape; the postorder walk destroys it safely.

Seven operations — **SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, INSERT and DELETE** — each walk one path, and each therefore costs $O(h)$. Deletion has three cases, distinguished by the number of children: detach a leaf; promote the only child; or, for two children, copy the inorder successor's key up and then delete the successor, which is guaranteed to be one of the two easy cases.

The whole structure lives or dies by h, and §6.13 showed that h can reach $n - 1$ on sorted input, costing a measured factor of about five hundred at $n = 16\,000$.

An **AVL tree** [3] fixes this by maintaining an invariant: the balance factor $\mathrm{bf}(x) = h(\mathrm{right}) - h(\mathrm{left})$ stays in $\{-1,0,+1\}$ at every node. Writing $N(h)$ for the fewest nodes an AVL tree of height h can hold gives

$$N(h) = N(h-1) + N(h-2) + 1, \qquad N(0) = 1,\ N(1) = 2,$$

whose solution is $N(h) + 1 = F(h+3)$ — a Fibonacci number — and whose consequence is

$$h \leq 1.4405\lg(n+2) - 1.3277 = O(\lg n).$$

An AVL tree is therefore never more than about 44% taller than a perfectly balanced tree [4].

The invariant is restored by **rotations**, which change a tree's shape while leaving its inorder walk untouched, and therefore always turn a binary search tree into a binary search tree. There are exactly **four cases** — Left-Left, Left-Right, Right-Right, Right-Left — identified by two balance factors and repaired by one or two rotations, summarized in **Table 6.5**.

Insertion performs at most one rebalancing, because the rotation restores the subtree to its pre-insertion height and no ancestor notices. **Deletion may perform one at every level**, because a rotation after a deletion can leave the subtree shorter than it was, and that shortening propagates. Both remain $O(\lg n)$.

Table 6.7 is the summary to memorize.

Table 6.7 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

Finally, §6.21 placed the AVL tree among its competitors. Against a **red-black tree** it is shorter and so searches faster, but rebalances more on deletion, which is why the C++ standard library's `std::map` is red-black [5] and why read-heavy workloads prefer AVL [6]. Against a **hash table** it is about ten times slower for single-key lookups and incomparably better for anything involving order: minimum, successor, sorted listing, range query.

6.24.1 Looking ahead

The next chapter leaves data structures and returns to algorithm design, moving from divide and conquer — which you met with Merge Sort in Chapter 3 — to **dynamic programming**. The bridge is a familiar face: the Fibonacci numbers that measured the thinnest AVL tree in §6.15 are also the standard first example of why naive recursion is exponential and why remembering answers makes it linear.

6.25 Exercises

6.1 Draw the binary search tree that results from inserting the keys 15,6,18,3,7,17,20,2,4,13,9 into an empty tree in that order. Then give its inorder, preorder and postorder walks.

6.2 For the tree of Exercise 6.1, give the result of SEARCH(13) as a path, SEARCH(8) as a path, MINIMUM, MAXIMUM, SUCCESSOR(13), SUCCESSOR(7) and PREDECESSOR(6).

6.3 A binary tree has preorder walk 50,30,20,40,70,60,80 and inorder walk 20,30,40,50,60,70,80. Reconstruct the tree. Then explain why the inorder walk alone would not have been enough.

6.4 Prove that if a node in a binary search tree has two children, then its inorder successor has no left child and its inorder predecessor has no right child.

6.5 Give a sequence of n distinct keys whose insertion into an empty binary search tree produces a tree of height $n - 1$, and a different sequence of the same n keys producing the minimum possible height. Do it concretely for $n = 7$ with the keys $1, \ldots, 7$.

6.6 Delete the key 15 from the tree of Exercise 6.1, using the inorder successor. Then start again from the original tree and delete 15 using the inorder predecessor. Are the two results the same tree?

6.7 Compute $N(h)$ for $h = 0,1, \ldots ,9$ from the recurrence $N(h) = N(h-1) + N(h-2) + 1$. What is the greatest height an AVL tree with 100 nodes can have? With 1000?

6.8 Verify the identity $N(h) + 1 = F(h+3)$ for $h = 7$ and $h = 8$ by computing both sides.

6.9 Insert the keys 10,20,30,40,50,25 one at a time into an empty AVL tree. After each insertion, state every balance factor, name the case if a rotation is needed, and draw the tree.

6.10 The tree in **Figure 6.8** is not an AVL tree. Identify z, y and x, name the case, perform the rotation or rotations, and draw the result with all balance factors.

6.11 Explain, with heights, why a single right rotation cannot repair a Left-Right imbalance. Give the resulting balance factors if one tries.

6.12 Prove that a rotation takes $\Theta(1)$ time regardless of subtree sizes, and that it leaves the inorder walk unchanged.

6.13 An AVL tree has height h. What are the smallest and largest number of nodes it may contain? Give both as formulas and evaluate them for $h = 5$.

6.14 Explain why an AVL *insertion* never needs more than one rebalancing operation, but an AVL *deletion* may need one at every level. Your answer must talk about the height of the rebalanced subtree.

6.15 You must store 10 million records and support: (a) look up by record ID; (b) list every record whose timestamp falls in a given hour; (c) find the record with the earliest timestamp. Which structure or structures would you use, and why? Justify each choice against **Table 6.6**.

6.16 Write a C++ function `int kthSmallest(Node* root, int k)` that returns the k-th smallest key in a binary search tree, 1-indexed, in $O(h + k)$ time without building a vector of all keys.

6.17 Modify **Listing 6.9** so that every node also stores the number of nodes in its subtree. Explain what must change in the rotations, and state which new operation this makes possible in $O(\lg n)$.

6.18 Prove that in any binary search tree the inorder walk visits the keys in increasing order, and use this to prove that the inorder successor of a node x with a right subtree is the minimum of that subtree.

6.26 Solutions to the Exercises

6.1 Insert one key at a time, always descending by comparison.

15 becomes the root. $6 < 15$, so it is the left child. $18 > 15$, right child. $3 < 15$, left to 6; $3 < 6$, left child of 6. $7 < 15$, left to 6; $7 > 6$, right child of 6. $17 > 15$, right to 18; $17 < 18$, left child of 18. $20 > 15$, right to 18; $20 > 18$, right child of 18. 2: left to 6, left to 3, left child of 3. 4: left to 6, left to 3, $4 > 3$, right child of 3. 13: left to 6, right to 7, $13 > 7$, right child of 7. 9: left to 6, right to 7, right to 13, $9 < 13$, left child of 13.

Writing the shape as `key{left,right}`:

```
15{6{3{2,4},7{.,13{9,.}}},18{17,20}}
```

The height is 4 (the path $15 \to 6 \to 7 \to 13 \to 9$). The three walks are

- inorder: 2,3,4,6,7,9,13,15,17,18,20 — sorted, as §6.6.1 promised;
- preorder: 15,6,3,2,4,7,13,9,18,17,20;
- postorder: 2,4,3,9,13,7,6,17,20,18,15.

6.2

SEARCH(13): 15 → 6 → 7 → 13. Found, four nodes visited.

SEARCH(8): $8 < 15$ go left; $8 > 6$ go right; $8 > 7$ go right; $8 < 13$ go left; $8 < 9$ go left, and the left child of 9 is empty. Path 15 → 6 → 7 → 13 → 9, failure. Had we been inserting 8, it would have become the left child of 9.

MINIMUM: 15 → 6 → 3 → 2, answer 2.

MAXIMUM: 15 → 18 → 20, answer 20.

SUCCESSOR(13): node 13 has no right child, so climb. 13 is the right child of 7 — keep climbing. 7 is the right child of 6 — keep climbing. 6 is the *left* child of 15 — stop. Answer 15, which is indeed what follows 13 in the inorder walk.

SUCCESSOR(7): node 7 has a right subtree, rooted at 13. Its minimum is found by walking left from 13, giving 9. Answer 9.

PREDECESSOR(6): node 6 has a left subtree, rooted at 3. Its maximum is found by walking right from 3, giving 4. Answer 4.

6.3 The first key of a preorder walk is always the root, so the root is 50. Find 50 in the inorder walk: it sits at position 4, with 20,30,40 before it and 60,70,80 after it. Everything before it is the left subtree, everything after it the right subtree — that is what "inorder" means.

The left subtree therefore holds {20,30,40} and the right holds {60,70,80}. Take the next three keys of the preorder walk, 30,20,40: these are the preorder walk of the left subtree, so its root is 30, its left subtree is {20} and its right is {40}. The remaining three preorder keys, 70,60,80, give the right subtree: root 70, left {60}, right {80}.

The reconstructed tree is exactly **Figure 6.1**.

The inorder walk alone is useless for reconstruction *precisely because* of the theorem in §6.6.1: every binary search tree on the same key set has the same inorder walk, namely the sorted list. The number of distinct binary-tree shapes on n keys is the Catalan number $C_n = \frac{1}{n+1}\binom{2n}{n}$, which for $n = 7$ is 429. The inorder walk cannot distinguish between those 429 trees; the preorder walk names the root at every level and so pins the shape down.

6.4 Let x have two children and let s be its inorder successor. By case 1 of §6.9, s is the minimum of x's right subtree, and TREE-MINIMUM stops exactly when the node it is standing on has no left child. Therefore s has no left child.

Symmetrically, the inorder predecessor p is the maximum of x's left subtree, and TREE-MAXIMUM stops exactly when the node has no right child, so p has no right child. ■

This is the fact that makes case 3 of deletion terminate: the node we recurse on has at most one child, so the recursion cannot meet case 3 a second time.

6.5 Any sorted sequence produces a chain. With $n = 7$ and keys $1, \ldots, 7$, inserting 1,2,3,4,5,6,7 sends every key right, giving a tree of height 6. So does the reverse, 7,6,5,4,3,2,1, which sends every key left.

For the minimum height, insert the median first and recurse: 4, then the medians of each half, 2 and 6, then the four remaining keys 1,3,5,7. The order 4,2,6,1,3,5,7 produces the perfect tree of height 2, which is the minimum permitted by Equation (6.1): $h \geq \lceil \lg(n+1) \rceil - 1 = \lceil \lg 8 \rceil - 1 = 2$.

6.6 With the **successor**: node 15 is the root and has two children. Its right subtree is rooted at 18; the minimum of that subtree is 17. Copy 17 into the root and delete the original node 17, which is a leaf. The result is

```
17{6{3{2,4},7{.,13{9,.}}},18{.,20}}
```

With the **predecessor**: the left subtree of 15 is rooted at 6; its maximum is found by walking right from 6 — to 7, then to 13 — giving 13. Node 13 has a left child 9 and no right child. Copy 13 into the root, then delete the original node 13, which is case 2: promote its only child 9. The result is

```
13{6{3{2,4},7{.,9}},18{17,20}}
```

The two results are **not** the same tree. Both are valid binary search trees holding the same ten keys, and both have the same inorder walk, 2,3,4,6,7,9,13,17,18,20 — but their shapes differ. Deletion is not uniquely determined; the choice of successor or predecessor is a free one, and implementations that always choose the successor tend to make the left side of the tree gradually heavier.

6.7 From $N(0) = 1$ and $N(1) = 2$:

$$N(2) = 4,\ N(3) = 7,\ N(4) = 12,\ N(5) = 20,\ N(6) = 33,\ N(7) = 54,\ N(8) = 88,\ N(9) = 143.$$

For 100 nodes: we need the largest h with $N(h) \leq 100$. Since $N(8) = 88 \leq 100$ and $N(9) = 143 > 100$, the answer is $h = 8$.

For 1000 nodes: continuing, $N(10) = 232$, $N(11) = 376$, $N(12) = 609$, $N(13) = 986$, $N(14) = 1596$. Since $N(13) = 986 \leq 1000 < 1596$, the answer is $h = 13$ — which is the value **Listing 6.6** printed.

6.8 $N(7) = 54$, so $N(7) + 1 = 55$. The Fibonacci numbers are $F(8) = 21, F(9) = 34, F(10) = 55$, so $F(7 + 3) = F(10) = 55$. ✓

$N(8) = 88$, so $N(8) + 1 = 89$, and $F(11) = F(10) + F(9) = 55 + 34 = 89$. ✓

6.9 Using $\text{bf} = h(\text{right}) - h(\text{left})$ throughout, and writing shapes as `key(bf){left,right}`:

Insert 10. `10(0)`. Height 0. No rotation.

Insert 20. Goes right of 10. `10(1){.,20(0)}`. Height 1. Balance factor $+1$ is legal; no rotation.

Insert 30. Right of 10, right of 20. Now $\text{bf}(20) = +1$ and $\text{bf}(10) = h(20) - h(\text{empty}) = 1 - (-1) = +2$: unbalanced. $z = 10$, $y = 20$ (right child), $x = 30$ (right child of y) — **Right-Right**. One left rotation about 10 gives `20(0){10(0),30(0)}`, height 1.

Insert 40. Right of 20, right of 30. $\text{bf}(30) = +1$, $\text{bf}(20) = h(30\text{-subtree}) - h(10) = 1 - 0 = +1$. Both legal. Tree: `20(1){10(0),30(1){.,40(0)}}`, height 2. No rotation.

Insert 50. Right of 20, right of 30, right of 40. Now $\text{bf}(40) = +1$, $\text{bf}(30) = h(40) - h(\text{empty}) = 1 - (-1) = +2$: unbalanced. $z = 30$, $y = 40$, $x = 50$ — **Right-Right**. One left rotation about 30 gives `20(1){10(0),40(0){30(0),50(0)}}`, height 2.

Insert 25. Path: $25 > 20$ go right to 40; $25 < 40$ go left to 30; $25 < 30$, attach as the left child of 30. Now $\text{bf}(30) = h(\text{empty}) - h(25) = -1 - 0 = -1$ and $h(30) = 1$; $\text{bf}(40) = h(50) - h(30) = 0 - 1 = -1$ and $h(40) = 2$; $\text{bf}(20) = h(40) - h(10) = 2 - 0 = +\mathbf{2}$: unbalanced. $z = 20$; $y = 40$, the right child; $x = 30$, the *left* child of y — **Right-Left**. First a right rotation about $y = 40$, then a left rotation about $z = 20$. The result is

```
30(0){20(0){10(0),25(0)},40(1){.,50(0)}}
```

height 2, and every balance factor legal. Six insertions, three rebalancings, and the tree never exceeded height 2 — whereas a plain binary search tree fed 10,20,30,40,50 would already be a chain of height 4.

6.10 The tree of **Figure 6.8** is: root 13; left child 9 with children 6 and 12; node 6 with children 5 and 8; node 5 with left child 3; right child 19 with left child 18.

Compute upward from 3: $h(3) = 0$; $h(5) = 1$ with $\text{bf}(5) = -1$; $h(8) = 0$; $h(6) = 2$ with $\text{bf}(6) = 0 - 1 = -1$; $h(12) = 0$; and $\text{bf}(9) = h(12) - h(6) = 0 - 2 = \mathbf{-2}$.

So $z = 9$, the deepest unbalanced node. The child of z on the path from 3 is $y = 6$, and the grandchild is $x = 5$. Node y is the *left* child of z and node x is the *left* child of y: this is the **Left-Left** case, confirmed by $\text{bf}(y) = -1 < 0$ in **Table 6.5**.

One right rotation about $z = 9$: node 6 moves up, 9 becomes 6's right child, and 6's old right child 8 becomes 9's left child. Attach 6 where 9 was, as the left child of 13. The result, with balance factors, is

```
13(-1){6(0){5(-1){3(0),.},9(0){8(0),12(0)}},19(-1){18(0),.}}
```

Note that node 13 was *also* out of range before the rotation ($\text{bf}(13) = 1 - 3 = -2$), and that repairing $z = 9$ fixed it automatically — because the rotation restored the subtree to its pre-insertion height, exactly as §6.17.1 and §6.20.2 explain.

6.11 Take the Left-Right configuration of **Figure 6.11**, with P_1 and P_4 of height k, node x of height $k + 1$, node y of height $k + 2$ and node z of height $k + 3$.

Perform a single right rotation about z. Node y moves to the top; P_1 — its old left subtree, height k — stays as its left subtree; z becomes its right child; and x, which was y's right subtree, is handed to z as z's left subtree.

Now z has children x (height $k + 1$) and P_4 (height k), so $h(z) = k + 2$. And y has children P_1 (height k) and z (height $k + 2$), so

$$\text{bf}(y) = (k + 2) - k = +2.$$

Still out of range, merely leaning the other way. A single rotation can only move one level of height across the root of the subtree; a zig-zag has the excess height buried one level deeper, so it must first be straightened into a zig-zig by the inner rotation. That is why LR and RL always cost two rotations.

6.12 *Constant time.* Inspect `rotateRight` in §6.16.1. It reads two pointers, writes two pointers, calls `refresh` twice — each of which reads two heights and writes one — and returns. That is a fixed number of machine operations, none of which depends on the number of nodes in P_1, P_2 or P_3: the subtrees are moved by changing the pointer that names them, never by visiting them. Hence $\Theta(1)$.

Inorder preserved. Before the rotation the subtree is y with left child x (subtrees P_1, P_2) and right subtree P_3. Its inorder walk is, by definition, the walk of x's subtree, then y, then P_3; and the walk of x's subtree is P_1, then x, then P_2. So the whole walk is

$$P_1,\ x,\ P_2,\ y,\ P_3.$$

After the rotation the subtree is x with left subtree P_1 and right child y (subtrees P_2, P_3). Its inorder walk is P_1, then x, then the walk of y's subtree, which is P_2, then y, then P_3 — giving

$$P_1,\ x,\ P_2,\ y,\ P_3,$$

the same sequence. Since a binary tree is a binary search tree exactly when its inorder walk is increasing, and the walk is unchanged, the rotated tree is still a binary search tree. ■

6.13 The **smallest** number of nodes is $N(h)$, given by the recurrence of §6.15: $N(h) = N(h-1) + N(h-2) + 1$, equivalently $N(h) = F(h+3) - 1$. The **largest** is achieved by the perfect binary tree, which fills every level: $1 + 2 + \cdots + 2^h = 2^{h+1} - 1$, and a perfect tree has every balance factor 0, so it is certainly an AVL tree.

For $h = 5$: the smallest is $N(5) = 20$ and the largest is $2^6 - 1 = 63$. So an AVL tree of height 5 holds between 20 and 63 nodes inclusive.

6.14 After an insertion that unbalances z, the subtree rooted at z has height $k + 3$ where $k + 2$ was its height before the insertion — see the height computations in §6.17.1 to §6.17.4. The rotation returns that subtree to height $k + 2$. Since the subtree's height is now identical to what it was before the insertion, every ancestor of z sees a child of unchanged height, so no ancestor's balance factor can have moved. One rebalancing suffices.

After a deletion the arithmetic differs by one. The deletion shortens some subtree; the rotation at z then produces a subtree that may be *one shorter* than before the deletion. That shortening is a genuine change of height as seen by z's parent, and it may push the parent's balance factor to ± 2. The same can happen at the grandparent, and so on. In the worst case the repair propagates to the root, one rebalancing per level, which is $O(\lg n)$ of them. **Listing 6.10** measures exactly this on Fibonacci trees: a constant 1 for insertion at every height, and 2,3,5,6,8 for deletion at heights 4,7,10,13,16.

6.15

(a) *Look up by record ID* is a pure point lookup: given an exact key, return the record. No ordering is involved. Use a **hash table**, expected $O(1)$. By the last row of **Table 6.6**, this is the case where a tree's extra cost buys nothing.

(b) *Every record whose timestamp falls in a given hour* is a range query. A hash table would have to scan all 10 million records; a balanced tree keyed by timestamp descends once and then walks only the matching records, $O(\lg n + k)$. Use an **AVL or red-black tree keyed by timestamp**. Whether AVL or red-black depends on the write rate: if the 10 million records are loaded and then mostly read, AVL's shorter height wins; if records arrive and are deleted continuously, red-black's bounded deletion cost wins.

(c) *The record with the earliest timestamp* is TREE-MINIMUM on the same timestamp-keyed tree: $O(\lg n)$, and free, because the index already exists.

The honest answer is therefore **two structures over the same data**: a hash index on record ID, and an ordered index on timestamp. This is exactly what a relational database does when you declare a hash index and a B-tree index on the same table, and it is why real systems maintain several indexes rather than one.

6.16 An inorder walk visits keys in increasing order, so the k-th key it emits is the k-th smallest. The trick is to stop the walk the moment the count reaches k, which makes the cost $O(h + k)$ rather than $\Theta(n)$: at most h nodes are traversed getting down to the smallest key, and at most k nodes are emitted after that.

Listing 6.14 The k-th smallest key of a binary search tree in $O(h+k)$ time. Compile with `g++ -std=c++17 -O2 -o kth kth.cpp`.

```
// Solution to Exercise 6.16: the k-th smallest key of a binary search tree,
// found by an inorder walk that stops as soon as it has counted k keys.
#include <iostream>
#include <stdexcept>

struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

Node* insertKey(Node* x, int k) {
    if (!x) return new Node(k);
    if      (k < x->key) x->left  = insertKey(x->left,  k);
    else if (k > x->key) x->right = insertKey(x->right, k);
    return x;
}

// Walks inorder but returns true the instant the k-th key is found, so the
// recursion unwinds without touching the rest of the tree.
bool kthHelper(Node* x, int& remaining, int& answer) {
    if (!x) return false;
    if (kthHelper(x->left, remaining, answer)) return true;   // found on the left
    if (--remaining == 0) { answer = x->key; return true; }   // this node is the k-th
    return kthHelper(x->right, remaining, answer);            // otherwise look right
}

int kthSmallest(Node* root, int k) {
    if (k < 1) throw std::out_of_range("k must be at least 1");
    int remaining = k, answer = 0;
    if (!kthHelper(root, remaining, answer))
        throw std::out_of_range("the tree has fewer than k keys");
    return answer;
}

int main() {
    Node* t = nullptr;
    for (int k : {15, 6, 18, 3, 7, 17, 20, 2, 4, 13, 9}) t = insertKey(t, k);

    for (int k = 1; k <= 11; ++k)
        std::cout << "k = " << k << "  ->  " << kthSmallest(t, k) << "\n";
```

Listing 6.14 (Continued) *The k-th smallest key of a binary search tree in O(h+k) time. Compile with g++ -std=c++17 -O2 -o kth kth.cpp.*

```
    try { kthSmallest(t, 12); }
    catch (const std::out_of_range& e) { std::cout << "k = 12 ->  " << e.what() << "\n"; }
    return 0;
}
```

Run on the tree of Exercise 6.1, **Listing 6.14** prints:

```
k = 1  ->  2
k = 2  ->  3
k = 3  ->  4
k = 4  ->  6
k = 5  ->  7
k = 6  ->  9
k = 7  ->  13
k = 8  ->  15
k = 9  ->  17
k = 10  ->  18
k = 11  ->  20
k = 12 ->  the tree has fewer than k keys
```

which is the inorder walk of Exercise 6.1, element by element.

6.17 Add a field `int size;` to `Node`, holding the number of nodes in the subtree rooted there, and maintain it exactly as the height is maintained: one new helper `sz`, which returns 0 for the empty tree, and one new line in `refresh`. **Listing 6.15** shows the whole result.

Listing 6.15 An order-statistic tree: the AVL tree of **Listing 6.9** with a subtree size in every node, supporting `select` and `rank` in $O(\lg n)$. Compile with `g++ -std=c++17 -O2 -o ostree ostree.cpp`.

```
// An AVL tree whose nodes also store their subtree sizes: an order-statistic tree.
// select(k) finds the k-th smallest key and rank(key) counts the smaller keys,
// both by a single descent, so both cost O(lg n).
#include <iostream>
#include <algorithm>

struct Node {
    int key, height, size;             // size = number of nodes in this subtree
    Node* left;
    Node* right;
    Node(int k) : key(k), height(0), size(1), left(nullptr), right(nullptr) {}
};

static int h (const Node* x) { return x ? x->height : -1; }
static int sz(const Node* x) { return x ? x->size   :  0; }
static int bf(const Node* x) { return h(x->right) - h(x->left); }
static void refresh(Node* x) {
    x->height = 1 + std::max(h(x->left), h(x->right));
    x->size   = 1 + sz(x->left) + sz(x->right);     // the new line
}

static Node* rotateRight(Node* y) {                 // unchanged from the AVL listing
    Node* x = y->left; y->left = x->right; x->right = y;
    refresh(y); refresh(x); return x;
}
static Node* rotateLeft(Node* x) {
    Node* y = x->right; x->right = y->left; y->left = x;
    refresh(x); refresh(y); return y;
}
static Node* rebalance(Node* z) {
    refresh(z);
    if (bf(z) < -1) {
        if (bf(z->left) > 0) z->left = rotateLeft(z->left);
        return rotateRight(z);
    }
    if (bf(z) > 1) {
        if (bf(z->right) < 0) z->right = rotateRight(z->right);
        return rotateLeft(z);
    }
    return z;
}
static Node* insert(Node* x, int key) {
    if (!x) return new Node(key);
    if      (key < x->key) x->left  = insert(x->left,  key);
    else if (key > x->key) x->right = insert(x->right, key);
    return rebalance(x);
```

Listing 6.15 (Continued) *An order-statistic tree: the AVL tree of Listing 6.9 with a subtree size in every node, supporting select and rank in O(\lg n). Compile with g++ -std=c++17 -O2 -o ostree ostree.cpp.*

```
}

// The k-th smallest key, 1-indexed. At each node L = size of the left subtree.
static int select(const Node* x, int k) {
    while (x) {
        int L = sz(x->left);
        if (k <= L)          x = x->left;          // the answer is on the left
        else if (k == L + 1) return x->key;        // exactly L keys are smaller
        else { k -= L + 1;   x = x->right; }       // skip the left subtree and x
    }
    return -1;                                      // k was out of range
}

// How many keys in the tree are smaller than key.
static int rank(const Node* x, int key) {
    int r = 0;
    while (x) {
        if (key <= x->key) x = x->left;
        else { r += sz(x->left) + 1; x = x->right; }
    }
    return r;
}

static void destroy(Node* x) { if (x) { destroy(x->left); destroy(x->right); delete x; } }

int main() {
    Node* root = nullptr;
    for (int k : {15, 6, 18, 3, 7, 17, 20, 2, 4, 13, 9}) root = insert(root, k);
    std::cout << "n = " << sz(root) << ", height = " << h(root) << "\n";
    std::cout << "select(k) for k = 1..11:";
    for (int k = 1; k <= sz(root); ++k) std::cout << " " << select(root, k);
    std::cout << "\n";
    for (int key : {2, 9, 15, 21})
        std::cout << "rank(" << key << ") = " << rank(root, key) << "\n";
    destroy(root);
    return 0;
}
```

Run on the keys of Exercise 6.1, it prints:

```
n = 11, height = 3
select(k) for k = 1..11: 2 3 4 6 7 9 13 15 17 18 20
rank(2) = 0
rank(9) = 5
rank(15) = 7
rank(21) = 11
```

Because both rotations already call `refresh` on the two nodes whose children changed — and in the correct bottom-up order — **no other change to the rotations is needed**. That is the payoff of having isolated the update in one helper. The insertion and deletion recursions also need no change, since every node on the path back up is passed to `rebalance`, which begins by calling `refresh`.

The new operation this makes possible is **order statistics in** $O(\lg n)$: the k-th smallest key can now be found by descending once, without the $O(k)$ walk of **Listing 6.14**; `select` in **Listing 6.15** is that descent. At a node x, let $L = \text{sz}(x\text{->left})$. If $k \le L$, recurse left; if $k = L + 1$, the answer is $\text{key}(x)$; otherwise recurse right looking for the $(k - L - 1)$-th smallest. The inverse operation, **rank** — "how many keys are smaller than k?" — comes free by the same descent, as `rank` shows: every time the descent turns right, the node and its whole left subtree are smaller, and their count

is added. A tree carrying subtree sizes is called an *order-statistic tree*, and it is the standard worked example of tree augmentation [7].

6.18 The first half is Theorem 6.6.1, proved there by induction on the number of nodes: the inorder walk of the empty tree is trivially increasing, and for a tree with root x the walk is (walk of L), key(x), (walk of R), whose three pieces are increasing by the induction hypothesis and whose two joints are increasing by the BST property at x.

For the second half, let x have a non-empty right subtree R, and let m be the minimum of R. Every key of R exceeds key(x), so every key of R is a candidate for "smallest key greater than key(x)", and key(m) is the smallest of them. It remains to rule out a candidate outside R. Let u be any node outside the subtree rooted at x with key(u) $>$ key(x). Walk down from the root toward x: at the last ancestor a where the path turned left into the subtree containing x, every key in a's right subtree — which is where u must lie — exceeds key(a), and key(a) exceeds every key in the subtree rooted at x, hence exceeds key(m). So key(u) $>$ key(m), and no outside node can beat m. Therefore the successor of x is m, the minimum of its right subtree. ■

References

[1] M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, 4th ed. Boston, MA, USA: Pearson, 2014.

[2] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[3] G. M. Adelson-Velsky and E. M. Landis, "An algorithm for the organization of information," *Doklady Akademii Nauk SSSR*, vol. 146, no. 2, pp. 263–266, 1962. (English translation: *Soviet Mathematics Doklady*, vol. 3, pp. 1259–1263, 1962.)

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

[5] ISO/IEC 14882:2024, *Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2024.

[6] B. Pfaff, "Performance analysis of BSTs in system software," in *Proc. ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems*, New York, NY, USA, 2004, pp. 410–411.

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

Chapter 7

From Divide and Conquer to Dynamic Programming

By the end of this chapter you will be able to look at a recursive algorithm and say, with evidence, whether it is wasting its life recomputing answers it has already found. You will be able to turn such an algorithm into one that is polynomial instead of exponential, by either of the two standard repairs — remembering answers as you go, or building them up from the smallest first — and you will know which of those two to reach for and why. You will be able to state what *optimal substructure* means, and prove that a particular problem has it. You will be able to take a chain of matrices, fill the dynamic-programming table by hand, and read the best way to bracket the product straight out of it. You will be able to compute the longest common subsequence of two strings, reconstruct the subsequence itself from a table of arrows, and cut the memory it needs from $\Theta(mn)$ down to $\Theta(\min(m,n))$ when you do not need the subsequence back. And you will be able to state, in one sentence you could defend before any skeptic, the difference between a problem that calls for divide and conquer and a problem that calls for dynamic programming.

7.1 Where We Left Off

Chapters 2 and 3 handed us a way to count and a way to argue. Chapter 3 in particular gave us the *recurrence*: an equation in which a function is defined in terms of itself at smaller inputs.

Recall the notation we fixed there, because this chapter leans on it constantly. If an algorithm on an input of size n breaks that input into a pieces, each of size n/b, solves each piece by calling itself, and spends $f(n)$ time doing everything else — the splitting beforehand and the joining afterward — then its running time $T(n)$ obeys Equation (7.1):

$$T(n) = aT(n/b) + f(n), \qquad a \geq 1,\ b > 1. \tag{7.1}$$

We called that the **general divide-and-conquer recurrence**, and we learned three ways to solve it: the *substitution method* (guess a bound, then prove it by induction), the *recursion-tree method* (draw the tree of calls, add up the cost of each level, then add up the levels), and the *Master Theorem* (a recipe that reads off the answer by comparing $f(n)$ against $n^{\log_b a}$).

We also met, in §3.8.9, the Fibonacci numbers, and computed them with a closed formula. In this chapter they come back as the villain of the opening act — and then as the hero of the repair.

Two pieces of vocabulary from Chapter 3 are worth restating before we go on, because the whole of this chapter is a statement about them.

> **DEFINITION.** A **subproblem** of a problem instance is a smaller instance of the *same* problem, produced by the algorithm in the course of solving the original. Computing F_5 is a subproblem of computing F_7; sorting $A[1..4]$ is a subproblem of sorting $A[1..8]$.
>
> **DEFINITION.** The **recursion tree** of a recursive call is the tree whose root is the original call, whose children are the calls that call makes directly, and so on down to the calls that make no

further calls (the *base cases*, drawn as leaves). Each node is labeled with the subproblem it solves.

Everything in this chapter follows from one observation about that tree: **sometimes the same label appears on more than one node.** When it does not, divide and conquer is the right tool. When it does — and when it does so catastrophically often — dynamic programming is the right tool. The rest of the chapter is spent making that sentence precise, proving it, measuring it, and applying it to three problems in full.

7.1.1 What is still missing

Chapter 3 assumed, without ever saying so, that the pieces a divide-and-conquer algorithm produces are *different pieces*. Merge Sort splits $A[1..8]$ into $A[1..4]$ and $A[5..8]$; those two halves share no element, and sorting one tells you nothing whatsoever about the other. Under that assumption the recursion tree has no repeated labels, every node does honest new work, and the Master Theorem's arithmetic is exactly right.

The assumption is not always true. When it fails, the Master Theorem still solves the recurrence correctly — it just solves it to an answer like $\Theta(1.618^n)$, which is a correct description of a catastrophe. This chapter is about noticing the catastrophe and preventing it.

7.2 Divide and Conquer as a Design Discipline

7.2.1 The three steps

Before we break divide and conquer, let us state it cleanly one final time. **Figure 7.1** is the statement in full.

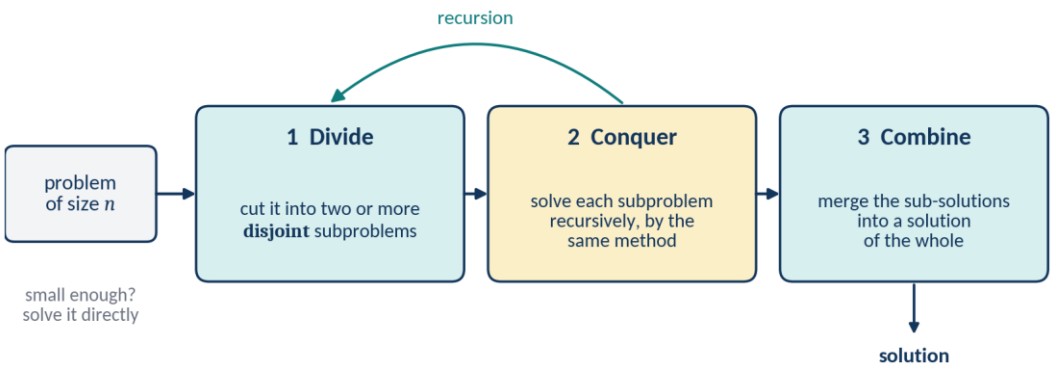


Figure 7.1 The divide-and-conquer method for algorithm design: divide the problem into **disjoint** subproblems, conquer them by recursion, then combine their solutions into a solution of the original.

Three words, each of which is a real instruction:

1. **Divide.** If the input is too large to handle in one go, cut it into two or more subproblems that are smaller instances of the same problem. The key word is *disjoint*: the pieces do not overlap.
2. **Conquer.** Solve each subproblem by calling the same procedure on it. If a subproblem is small enough — usually size 0 or 1 — solve it directly, without recursion. That direct case is the **base case**, and without one the recursion never stops.
3. **Combine.** Take the solutions of the subproblems and merge them into a solution of the original problem.

IN PLAIN LANGUAGE. You have a mountain of laundry. You cannot wash it all at once, so you split it into two piles (divide), wash each pile the same way you would have washed the whole

> (conquer), and then put the two clean piles together into one clean mountain (combine). The two piles are *disjoint*: no shirt is in both. Washing the first pile teaches you nothing about the second, and you would not dream of washing the same shirt twice.

7.2.2 Merge Sort, seen once more as a shape

The standard example is Merge Sort, and **Figure 7.2** pictures it.

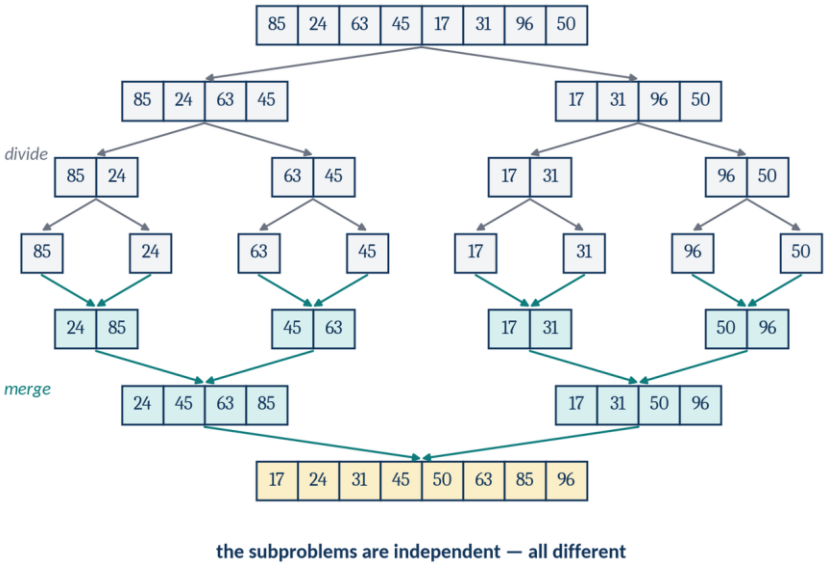


Figure 7.2 Merge Sort as divide and conquer. The array is split in half, each half is sorted recursively, and the two sorted halves are merged. The sentence that matters is written beneath the drawing: **the subproblems are independent, all different.**

Read that sentence in **Figure 7.2** slowly, because it is the hinge of this entire chapter: *the subproblems are independent, all different*. When Merge Sort is asked to sort a sixteen-element array, it asks for two eight-element arrays. Those two never coincide. Each of them asks for two four-element arrays; those four never coincide either. No label is ever repeated anywhere in the tree. Every node does work that no other node does.

That is why the recurrence $T(n) = 2T(n/2) + \Theta(n)$ tells the whole truth about Merge Sort, and why the answer $\Theta(n\lg n)$ is a fast algorithm rather than a slow one.

7.2.3 When divide and conquer is the right tool

Divide and conquer earns its keep exactly when three conditions hold together:

- the problem genuinely splits — there is a natural way to cut the input into smaller instances of the same problem;
- the pieces are **disjoint**, so that no work is duplicated across them;
- the **combine** step is cheap relative to the work below it, so that the recursion tree's levels do not all cost more than the last.

Merge Sort satisfies all three. So does binary search (one piece instead of two, combine cost zero, $T(n) = T(n/2) + \Theta(1) = \Theta(\lg n)$). So does Strassen's matrix multiplication ($T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{\lg 7})$). So does Quicksort, on average.

7.2.4 When divide and conquer wastes work

Now suppose the second condition fails. Suppose the pieces *overlap* — that the subproblem "sort $A[1..4]$" and the subproblem "sort $A[3..6]$" both show up, sharing two elements; or, worse, that the very same subproblem shows up twice, three times, a thousand times.

A plain recursive implementation has no memory. It does not know it has seen a subproblem before. Ask it for the answer twice and it will compute the answer twice, from scratch, all the way down to the leaves. Ask it a million times and it will do a million full computations.

> **THE TRAP.** Recursion is not slow. *Recomputation* is slow. A recursive procedure whose subproblems never repeat is as fast as anything; a recursive procedure whose subproblems repeat is an engine for doing the same work over and over, and its cost can be exponential even when the number of genuinely different questions it needs answered is tiny.

How bad can it get? The smallest interesting example — three lines of code, no cleverness, taught to beginners everywhere — is bad enough to be unusable. That example is Fibonacci, and it is where we go next.

7.3 Fibonacci: the Smallest Problem That Breaks Divide and Conquer

7.3.1 The definition

Figure 7.3 states the definition.

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad (n \geq 2)$$

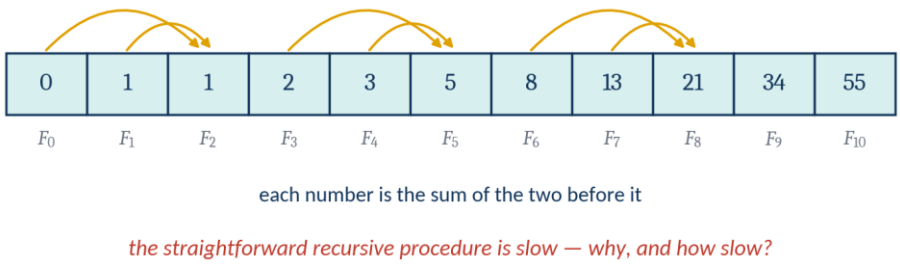


Figure 7.3 The Fibonacci numbers, with the warning that follows immediately: *the straightforward recursive procedure is slow*. Why, and how slow, is the business of the next four sections.

In symbols, and introducing each one as we go: we write F_n (read "F sub n") for the n-th Fibonacci number, where n is a whole number 0,1,2,3, … called the **index**. The definition is Equation (7.2):

$$F_0 = 0, \qquad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2. \tag{7.2}$$

The first two lines are the **base cases**: they state outright what F_0 and F_1 are, with no computation. The third line is the **recursive case**: it says that from the third number onwards, each number is the sum of the two before it.

Let us produce the sequence by hand, one step per line, so that nothing is taken on trust:

- $F_0 = 0$ (given)
- $F_1 = 1$ (given)
- $F_2 = F_1 + F_0 = 1 + 0 = 1$
- $F_3 = F_2 + F_1 = 1 + 1 = 2$

- $F_4 = F_3 + F_2 = 2 + 1 = 3$
- $F_5 = F_4 + F_3 = 3 + 2 = 5$
- $F_6 = F_5 + F_4 = 5 + 3 = 8$
- $F_7 = F_6 + F_5 = 8 + 5 = 13$
- $F_8 = F_7 + F_6 = 13 + 8 = 21$
- $F_9 = F_8 + F_7 = 21 + 13 = 34$
- $F_{10} = F_9 + F_8 = 34 + 21 = 55$

which is the sequence 0,1,1,2,3,5,8,13,21,34,55, … printed in **Figure 7.3**.

7.3.2 The straightforward recursion

The definition translates into code so directly that **Pseudocode 7.1** is almost a transcription:

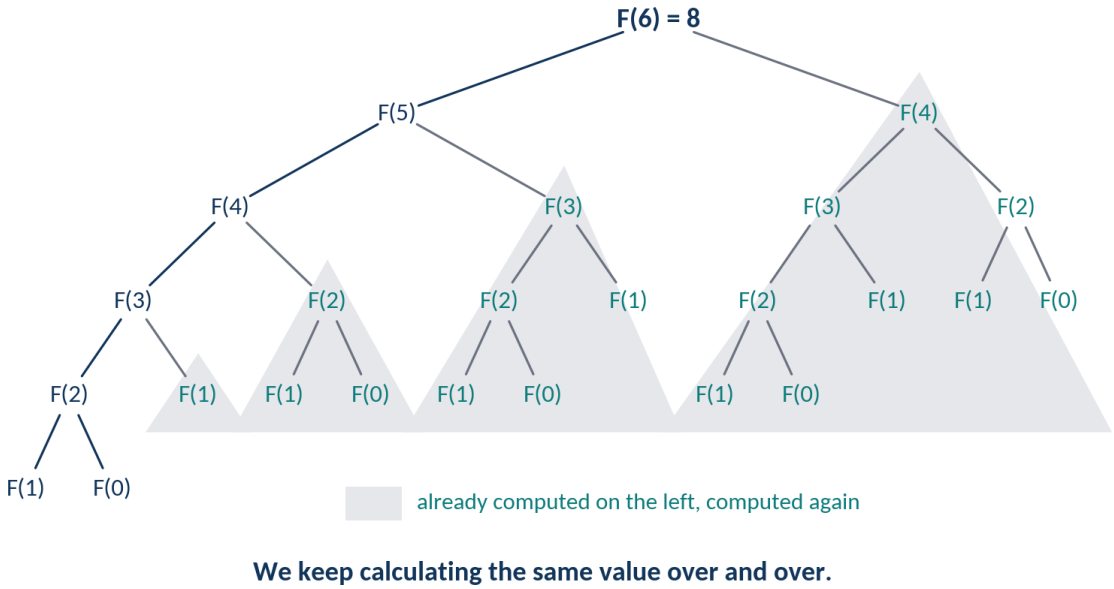
Pseudocode 7.1 The straightforward recursion for F_n, transcribed directly from the definition.

```
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
}
```

It is correct. It is also, as **Figure 7.3** warns, slow — and not "a bit slow", but slow in the way that makes a program useless. To see why, we draw the recursion tree.

7.3.3 The recursion tree

Figure 7.4 draws the recursion tree for F_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.

Walk through it. The root asks for $F(6)$. To answer, it must ask for $F(5)$ and $F(4)$. The $F(5)$ call must ask for $F(4)$ and $F(3)$ — and there is the first duplication: $F(4)$ is now going to be computed twice, once under $F(5)$ and once directly under the root. Neither call knows about the other. Each will descend all the way to the leaves.

Go one level deeper and it gets worse. $F(3)$ appears three times. $F(2)$ appears five times. $F(1)$ appears eight times. The sentence printed under the drawing is the whole diagnosis: **we keep calculating the same value over and over.**

7.3.4 Counting the calls exactly

"Over and over" is a feeling. Let us make it a number.

Let $C(n)$ be the total number of invocations of `fib` made when you call `fib(n)` — counting the outermost call itself. From the code:

- If $n \leq 1$ the function returns immediately, making no further calls. So $C(0) = 1$ and $C(1) = 1$.
- Otherwise the call makes one invocation of its own, plus everything `fib(n-1)` does, plus everything `fib(n-2)` does. So $C(n) = C(n-1) + C(n-2) + 1$.

That is itself a recurrence, and it has a beautiful closed form.

> **CLAIM.** $C(n) = 2F_{n+1} - 1$ for every $n \geq 0$.

Proof, by strong induction on n.

Base cases. For $n = 0$: $2F_1 - 1 = 2(1) - 1 = 1 = C(0)$. ✓ For $n = 1$: $2F_2 - 1 = 2(1) - 1 = 1 = C(1)$. ✓

Inductive step. Let $n \geq 2$ and suppose the claim holds for all smaller indices. Then

$$C(n) = C(n-1) + C(n-2) + 1 = (2F_n - 1) + (2F_{n-1} - 1) + 1.$$

Collecting terms, the three constants $-1, -1, +1$ sum to -1, and the two doubled Fibonacci terms factor:

$$C(n) = 2(F_n + F_{n-1}) - 1 = 2F_{n+1} - 1,$$

where the last step is just the Fibonacci recurrence read forward. ■

Two companion counts fall out of the same picture, and we will want both:

- The **leaves** of the tree are exactly the calls with $n \leq 1$. Let $L(n)$ be their number. Then $L(0) = L(1) = 1$ and $L(n) = L(n-1) + L(n-2)$ — the Fibonacci recurrence itself, with the indices shifted — so $L(n) = F_{n+1}$.
- The **internal nodes** are the calls that perform an addition. There are $C(n) - L(n) = (2F_{n+1} - 1) - F_{n+1} = F_{n+1} - 1$ of them. That is the answer to the question "how many summations are there?".

For $n = 6$ these give $C(6) = 2F_7 - 1 = 2(13) - 1 = 25$ calls, $L(6) = F_7 = 13$ leaves, and $F_7 - 1 = 12$ additions. Count the nodes in **Figure 7.4** if you like; there are twenty-five.

7.3.5 Why the cost is $\Theta(\phi^n)$

We now have $C(n) = 2F_{n+1} - 1$, so the running time of the naive procedure is $\Theta(F_n)$. That only helps if we know how fast F_n itself grows. **Figure 7.5** gives the answer in outline.

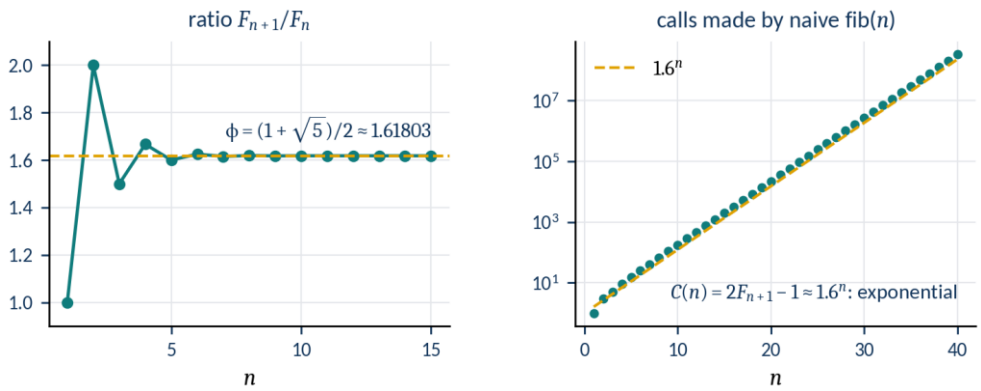


Figure 7.5 The argument in outline. Left: consecutive Fibonacci numbers approach the golden ratio $\phi \approx 1.61803$, so $F_n \approx 1.6^n$. Right: the call count $2F_{n+1} - 1$ of the naive procedure, on a logarithmic scale, runs parallel to 1.6^n — the tree has about 1.6^n leaves, and the running time is exponential.

Let us derive that properly rather than accept it. Introduce the symbol ϕ (Greek letter *phi*), defined as

$$\phi = \frac{1+\sqrt{5}}{2} = 1.6180339887\ldots$$

and its companion ψ (*psi*), defined as

$$\psi = \frac{1-\sqrt{5}}{2} = -0.6180339887\ldots$$

Both are roots of the quadratic $x^2 = x + 1$; you can check this by substitution. That single fact is what ties them to Fibonacci, because the Fibonacci recurrence $F_n = F_{n-1} + F_{n-2}$ has exactly the same shape: if you try a solution of the form $F_n = x^n$, you get $x^n = x^{n-1} + x^{n-2}$, and dividing through by x^{n-2} gives $x^2 = x + 1$.

Because both ϕ^n and ψ^n solve the recurrence, so does any combination $A\phi^n + B\psi^n$. Choosing A and B to make the base cases come out right ($F_0 = 0, F_1 = 1$) gives $A = 1/\sqrt{5}$ and $B = -1/\sqrt{5}$, and hence **Binet's formula**, Equation (7.3):

$$F_n = \frac{\phi^n - \psi^n}{\sqrt{5}}. \tag{7.3}$$

Now the key observation about Equation (7.3). $|\psi| = 0.618\ldots < 1$, so ψ^n shrinks toward zero as n grows; in fact $\left|\psi^n/\sqrt{5}\right| < 1/2$ for every $n \geq 0$. Therefore

$$F_n = \frac{\phi^n}{\sqrt{5}} + \varepsilon_n, \qquad |\varepsilon_n| < 1/2,$$

which says F_n is the nearest whole number to $\phi^n/\sqrt{5}$. In asymptotic language, $F_n = \Theta(\phi^n)$. Knuth [1, §1.2.8] develops the Fibonacci numbers and the golden ratio at length, including the history of Binet's formula.

Putting the pieces together gives Equation (7.4):

$$C(n) = 2F_{n+1} - 1 = \Theta(\phi^n) = \Theta(1.618^n). \tag{7.4}$$

> **THE VERDICT.** The straightforward recursion runs in $\Theta(\phi^n)$ time — exponential. Adding one to n multiplies the work by about 1.618; adding two multiplies it by $\phi^2 = 2.618$. This is not a constant-factor problem that a faster computer fixes. A computer one thousand times faster buys you about fourteen extra values of n.

7.3.6 How much work is actually repeated

Here is the part that should make you angry on the algorithm's behalf. To compute F_n there are only $n + 1$ different questions to answer: what is F_0, what is F_1, …, what is F_n. That is it. Yet the naive procedure asks $2F_{n+1} - 1$ questions.

How many times is each individual value recomputed? Reading **Figure 7.4**, and confirmed by the program of §7.3.7, the number of calls to $F(k)$ inside `fib(n)` is given by Equation (7.5):

$$\text{calls to } F(k) \;=\; F_{n-k+1} \quad (k \geq 1), \qquad \text{calls to } F(0) \;=\; F_{n-1}. \tag{7.5}$$

For $n = 6$ that predicts $F(6)$ once, $F(5)$ once, $F(4)$ twice, $F(3)$ three times, $F(2)$ five times, $F(1)$ eight times and $F(0)$ five times — note that those multiplicities are themselves the Fibonacci numbers, read backward. Twenty-five calls to answer seven distinct questions.

7.3.7 Measuring it

Numbers on a page are one thing; a program that will not finish is another. **Listing 7.1** instruments the naive recursion: it counts every call, records how many times each $F(k)$ is asked for, and times the whole thing.

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`.

```
// 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;
}
```

Compiled and run, it prints:

```
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
```

Every number we predicted by hand is there. The call profile of `fib(6)` is exactly 25 calls split as 1,1,2,3,5,8,5. And look at the last column: the ratio of successive call counts, two steps apart, is 2.618 to three decimal places — which is $\phi^2 = 2.6180339\ldots$, precisely as the theory demands.

The practical consequence is in the timing column. Computing F_{42} — a number that fits comfortably in a 32-bit integer, a number you could write on a napkin in five seconds — takes nearly two seconds and eight hundred and sixty-six million function calls. Extrapolating at ϕ per step, F_{90} would take this program about $1.94 \times \phi^{48} \approx 2 \times 10^{10}$ seconds — roughly seven centuries.

> **DO NOT MISREAD THIS.** The problem is not that the algorithm is recursive, and it is not that Fibonacci numbers are hard. The problem is that the recursion tree contains $2F_{n+1} - 1$ nodes carrying only $n + 1$ distinct labels. Everything else in this chapter is a technique for collapsing those duplicate nodes into one.

7.4 Four Repairs to Fibonacci

The prescription is on **Figure 7.6**, and it contains, compressed into one picture, the whole idea of dynamic programming.

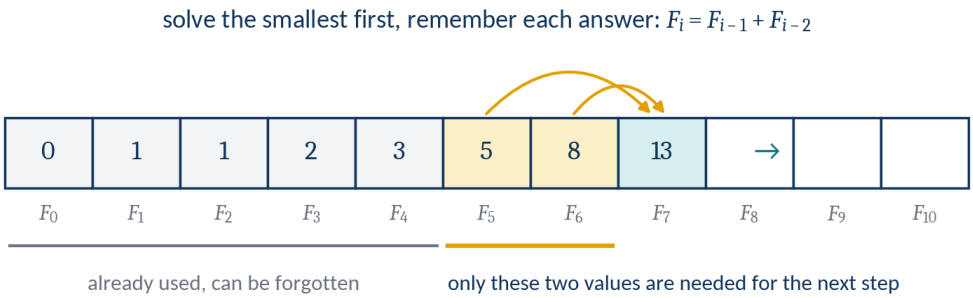


Figure 7.6 The repair, in brief: compute F_n in linear time by *remembering solutions to the solved subproblems* — dynamic programming — building the solution bottom-up, trading space for time. The highlighted pair is the observation that will become §7.4.3: only two values need to be remembered at any time.

The technique has a name and an inventor: Richard Bellman coined *dynamic programming* in the 1950s [2], choosing the word "programming" in its older sense of *planning a schedule of actions*, not in the sense of writing code.

Notice the phrase **trade space for time**. That is the bargain being struck. We will spend memory we did not previously spend, and in exchange we will stop paying for work we have already done.

In the Fibonacci case the bargain is spectacularly good: we spend $\Theta(n)$ words of memory (and then, with a little thought, only two words), and we go from $\Theta(\phi^n)$ time to $\Theta(n)$.

We will make the repair four different ways. The first two are the two canonical forms of dynamic programming, and everything later in this chapter is one or the other of them. The third is an optimization of the second. The fourth is a bonus that leaves dynamic programming behind entirely and is included because it is beautiful, and because it shows that even a linear-time answer can be beaten.

7.4.1 Repair 1: memoization — remember what you have already answered

> **DEFINITION. Memoization** is the technique of keeping a table of answers already computed, consulting it at the top of every recursive call, and returning the stored answer instead of recursing when one is found. The recursion's *structure* is unchanged; only its *repetition* is removed. (The word is "memoization", from *memo*, a note to oneself — not "memorization".)

The change to the code is three lines, shown in **Pseudocode 7.2**:

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;
}
```

Why is this linear? Because of an argument you should learn to make automatically.

Each table entry `memo[k]` is written exactly once, the first time $F(k)$ is genuinely computed. Every subsequent request for $F(k)$ takes the first branch and returns in constant time. There are $n+1$ entries. Each of the $n+1$ genuine computations does $O(1)$ work of its own (one addition, one store), because the two recursive calls it makes are *charged to their own entries*, not to this one. Hence the total is $\Theta(n)$ genuine computations plus $\Theta(n)$ lookups, that is, $\Theta(n)$ time and $\Theta(n)$ space.

> **THE ACCOUNTING TRICK.** When analyzing a memoized recursion, do not try to follow the calls. Instead count the table entries and multiply by the cost of filling one entry, *ignoring* the recursive calls inside it (they are paid for by their own entries). Here: $n+1$ entries $\times$ $O(1)$ each $=\Theta(n)$. This rule works for every memoized algorithm in this chapter.

In the recursion tree of **Figure 7.4**, memoization amounts to shading every gray triangle out of existence: the first time a subtree is entered it is computed, and every later arrival at the same label stops at the root of that subtree and reads the answer off.

7.4.2 Repair 2: bottom-up — build the small answers first

Memoization is recursion that has learned not to repeat itself. The **bottom-up** method abandons recursion altogether. It notices that if we are going to compute all of $F_0, F_1, \ldots, F_n$ anyway, we may as well compute them in the order $F_0, F_1, F_2, \ldots$, in which case every value we need is already sitting in the table when we reach for it. **Pseudocode 7.3** does exactly that.

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];
}
```

One loop, $n - 1$ iterations, one addition each: $\Theta(n)$ time, $\Theta(n)$ space, no recursion, no stack.

The ordering is the whole art. We must arrange the loop so that when the algorithm computes an entry, everything that entry depends on has already been computed. Here that is easy — F_i depends on F_{i-1} and F_{i-2}, both of which have smaller index, so increasing i works. In §7.6 the correct order will be much less obvious, and getting it right will be the crux of the algorithm.

7.4.3 Repair 3: $\Theta(1)$ space

Look again at the two highlighted cells of **Figure 7.6**: *in this case, only two values need to be remembered at any time.*

That is the second great economy of dynamic programming, and it is worth naming. **Pseudocode 7.4** puts it into practice.

> **THE SPACE ECONOMY.** If the recurrence only ever reaches back a bounded distance, you do not need the whole table — only a sliding window of the entries still in reach. Everything older can be forgotten. F_i reaches back exactly two places, so a window of two suffices.

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;
}
```

$\Theta(n)$ time, $\Theta(1)$ space. Trace it for $n = 6$, one line per iteration, so that the sliding is visible:

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

The same trick, applied to a two-dimensional table instead of a one-dimensional one, is exactly what gives the $\Theta(\min(m, n))$-space version of the longest common subsequence in §7.9.10. Learn it here where it is easy.

7.4.4 Repair 4 (bonus): the matrix power, $O(\lg n)$

All three repairs above are $\Theta(n)$, and for a long time people assumed that was the end of the story: you cannot compute the n-th term of a sequence in less than n steps. You can.

Start from the observation that one step of the Fibonacci recurrence is a linear map. Write the state as a column of two consecutive values. Then

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}.$$

Check the arithmetic of that claim: the top row of the product is $1 \cdot F_n + 1 \cdot F_{n-1} = F_{n+1}$ ✓, and the bottom row is $1 \cdot F_n + 0 \cdot F_{n-1} = F_n$ ✓.

Call that matrix $Q = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. Applying the map n times from the starting state $(F_1, F_0)^\top = (1,0)^\top$ gives

$$Q^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}, \tag{7.6}$$

Equation (7.6), a fact proved by an easy induction ($n = 1$ is Q itself; the step is one matrix multiplication and three uses of $F_k = F_{k-1} + F_{k-2}$). So F_n is the top-right entry of Q^n.

Now, how fast can we raise a matrix to the n-th power? Not by multiplying n times — that would be $\Theta(n)$ again. By **binary exponentiation**, also called *repeated squaring*:

$$Q^n = \begin{cases} I & n = 0, \\ (Q^{n/2})^2 & n \text{ even}, \\ Q \cdot Q^{n-1} & n \text{ odd}. \end{cases}$$

Each halving of n costs one 2×2 matrix multiplication for the squaring, plus one more when n is odd — at most two, and each is eight scalar multiplications and four additions, a constant. The number of halvings before n reaches zero is $\lfloor \lg n \rfloor + 1$. Hence $O(\lg n)$ arithmetic operations in total.

> **THE SMALL PRINT.** $O(\lg n)$ counts *arithmetic operations*, treating each addition and multiplication as unit cost — the RAM model of Chapter 2. F_n has about $0.209n$ decimal digits (about $0.694n$ bits), so for very large n the numbers themselves stop fitting in a machine word and each operation stops being constant-time. The program below sidesteps this by working modulo 1 000 000 007, a common convention that keeps every value inside a 64-bit integer. If you need the exact value of $F_{1\,000\,000}$, all four methods must use big integers, and the comparison changes.

7.4.5 All four, measured

Listing 7.2 implements all four repairs and times them on the same inputs.

Listing 7.2 Memoized, bottom-up, constant-space and matrix-power Fibonacci, all four timed against each other. Compile with `g++ -std=c++17 -O2 -o fib_fast fib_fast.cpp`.

```cpp
// fib_fast.cpp - four ways to compute F(n) that are not exponential.
// Values are taken modulo 1000000007 so that they stay inside a 64-bit integer
// for large n; the running times are unaffected by that choice.
#include <iostream>
#include <iomanip>
#include <vector>
#include <array>
#include <chrono>
using namespace std;
using namespace std::chrono;

const unsigned long long MOD = 1000000007ULL;
const int MEMO_LIMIT = 250000;   // measured: deeper recursion overflows an 8 MiB stack

// (1) Top-down with memoization.
vector<long long> memo;
long long fibMemo(int n) {
    if (memo[n] != -1) return memo[n];           // already solved: look it up, do not
recurse
    long long r = (n <= 1) ? n : (fibMemo(n - 1) + fibMemo(n - 2)) % MOD;
    memo[n] = r;                                 // record the answer before returning it
    return r;
}

// (2) Bottom-up over a full table.
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]) % MOD;
    return F[n];
}

// (3) Bottom-up keeping only two values: Theta(1) space.
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) % MOD;
        prev = cur; cur = next;
    }
    return cur;
}
```

Listing 7.2 (Continued) *Memoized, bottom-up, constant-space and matrix-power Fibonacci, all four timed against each other. Compile with g++ -std=c++17 -O2 -o fib_fast fib_fast.cpp.*

```
// (4) Matrix power: [[1,1],[1,0]]^n holds F(n) in position (0,1).
typedef array<array<unsigned long long,2>,2> Mat;
Mat mul(const Mat& A, const Mat& B) {
    Mat C{};
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j) {
            unsigned long long s = 0;
            for (int k = 0; k < 2; ++k) s = (s + A[i][k] * B[k][j]) % MOD;
            C[i][j] = s;
        }
    return C;
}
long long fibMatrix(int n) {
    Mat R{{{{1,0}},{{0,1}}}};                    // identity
    Mat B{{{{1,1}},{{1,0}}}};
    int e = n;
    while (e > 0) {                              // binary exponentiation: about lg n
squarings
        if (e & 1) R = mul(R, B);
        B = mul(B, B);
        e >>= 1;
    }
    return (long long)R[0][1];
}

template <class F> double timeIt(F f, int n, int reps, long long& out) {
    auto t0 = steady_clock::now();
    for (int r = 0; r < reps; ++r) out = f(n);
    auto t1 = steady_clock::now();
    return duration<double, micro>(t1 - t0).count() / reps;
}

int main() {
    cout << "F(n) mod 1000000007, and microseconds per call\n\n";
    cout << left << setw(10) << "n" << setw(14) << "F(n)"
         << setw(13) << "memo us" << setw(13) << "table us"
         << setw(13) << "iter us" << setw(13) << "matrix us" << "\n";
    for (int n : {40, 1000, 10000, 100000, 250000, 1000000}) {
        long long a = -1, b, c, d;
        double tm = -1;
        if (n <= MEMO_LIMIT) { memo.assign(n + 1, -1);
            tm = timeIt([](int k){ return fibMemo(k); }, n, 1, a); }
        double tt = timeIt(fibTable,  n, 5, b);
        double ti = timeIt(fibIter,   n, 5, c);
        double tx = timeIt(fibMatrix, n, 5, d);
        if (!(b == c && c == d) || (a != -1 && a != b)) { cout << "MISMATCH\n"; return 1; }
        cout << left << setw(10) << n << setw(14) << b;
```

Listing 7.2 (Continued) *Memoized, bottom-up, constant-space and matrix-power Fibonacci, all four timed against each other. Compile with g++ -std=c++17 -O2 -o fib_fast fib_fast.cpp.*

```
        if (tm >= 0) cout << setw(13) << fixed << setprecision(2) << tm;
        else         cout << setw(13) << "overflow";
        cout << setw(13) << fixed << setprecision(2) << tt
             << setw(13) << ti << setw(13) << tx << "\n";
    }
    cout << "\nAll methods that ran agreed on every n above.\n";
    cout << "F(0..10) exactly: 0";
    long long p = 0, q = 1;
    for (int i = 1; i <= 10; ++i) { cout << ", " << q; long long t = p + q; p = q; q = t; }
    cout << "\n";
    return 0;
}
```

Its output:

```
F(n) mod 1000000007, and microseconds per call

n         F(n)          memo us      table us     iter us      matrix us
40        102334155     0.83         0.53         0.10         0.03
1000      517691607     18.34        4.04         3.19         0.05
10000     271496360     177.21       45.30        32.21        0.07
100000    911435502     2140.96      523.36       325.41       0.09
250000    608672132     4561.73      1259.70      821.79       0.11
1000000   918091266     overflow     5041.49      3378.51      0.13

All methods that ran agreed on every n above.
F(0..10) exactly: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

Six things in that table deserve comment, and together they are the practical content of this chapter.

First, compare against **Listing 7.1**. The naive recursion needed 735 490 microseconds for $n = 40$. The memoized version needs 0.83. That is a speed-up of about 886 000 times, from adding two lines of code.

Second, the three $\Theta(n)$ methods really are linear. Multiply n by ten, from 10 000 to 100 000, and `table` goes from 45.30 to 523.36 microseconds — a factor of 11.6, close to the factor of 10 that linearity predicts, with the excess explained by the memory traffic of a ten-times-larger array.

Third, memoization is consistently the slowest of the linear three — 2141 microseconds against 523 for the table, four times slower at $n = 100\,000$. The work done is the same. The difference is overhead: a function call, a stack frame, a comparison against the sentinel and a return, per entry, instead of one loop iteration per entry.

Fourth, the constant-space iteration beats the full table (325 against 523 microseconds at $n = 100\,000$) even though both are $\Theta(n)$ and do the same additions. It never allocates, never touches memory outside three registers, and never suffers a cache miss. Asymptotics say they are equal; the machine says otherwise. Both statements are true.

Fifth, the memoized column says `overflow` at $n = 1\,000\,000$, and that is not a typo. The recursion reaches depth n. With the default 8 MiB stack on the machine used here, it survived $n = 250\,000$ and crashed with a segmentation fault at $n = 300\,000$. Top-down recursion has a hidden resource cost that bottom-up iteration does not.

Sixth, and most striking: the matrix method took 0.13 microseconds for $n = 1\,000\,000$ — about twenty-six thousand times faster than the iterative version. Its column barely moves at all as n grows by four orders of magnitude, from 0.03 to 0.13 microseconds, because the work is proportional to $\lg n$, and lg of a million is only twenty.

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

7.5 Optimization Problems

Fibonacci was a warm-up. It has repeated subproblems, but it is not an *optimization* problem: there is nothing to choose, because F_n has exactly one value. The problems for which dynamic programming was invented all involve choosing. **Figure 7.7** frames them.

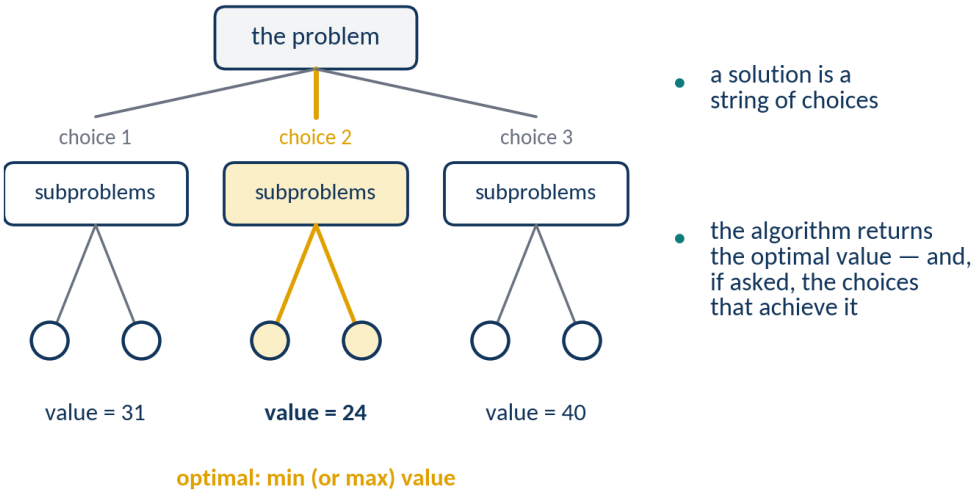


Figure 7.7 Optimization problems in outline: many solutions exist, one is optimal, a solution is a *string of choices*, and the algorithm computes the optimal value and — if we ask for it — the optimal solution itself.

7.5.1 What makes a problem an optimization problem

> **DEFINITION.** An **optimization problem** is one in which each input admits many **feasible solutions**, each feasible solution carries a numerical **value** (a cost, a length, a profit), and the task is to find a feasible solution whose value is smallest (**minimization**) or largest (**maximization**). Such a solution is called **an optimal solution** — "an", not "the", because several different solutions may tie at the optimal value.

Three things in that framing deserve emphasis.

A solution exhibits a structure. It is not an atom. It is built.

It consists of a string of choices that were made. If we can name the choices, and if each choice reduces the problem to one or more smaller problems of the same kind, then the solution is a *path* through a space of subproblems — and a path is something we can search systematically.

The algorithm computes the optimal value plus, if needed, the optimal solution. These are two different outputs and two different amounts of work. The value is a number; the solution is the object achieving it. Dynamic programming naturally produces the value, and produces the solution only if we additionally record, at each step, *which* choice attained the optimum. That extra record is the *c* table of §7.6.9 and the *b* table of §7.9.8.

7.5.2 Optimal substructure, stated precisely

The single property that makes dynamic programming possible is **optimal substructure**. It is easy to state loosely and easy to get wrong, so here it is carefully.

> **DEFINITION.** A problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to the subproblems it is composed of. Formally: if S is an optimal solution to instance P, and S decomposes into solutions $S_1, \dots, S_r$ of subproblems $P_1, \dots, P_r$, then each S_i must itself be an optimal solution to P_i.

Note precisely what is and is not being claimed.

- It **is** claimed that if you take an optimal solution apart, the pieces are optimal for their own subproblems.
- It is **not** claimed that gluing optimal pieces together gives an optimal whole. That is false in general: optimal pieces for the *wrong* decomposition glue into rubbish. This is why the algorithms in this chapter try *every* decomposition and take the best.

Optimal substructure is what licenses the recurrence. If you know that the optimal solution splits at some point k, and you know that both sides are optimal for their halves, then the optimal value is $\min_k\{\text{best-left}(k) + \text{best-right}(k) + \text{cost of splitting at } k\}$ — and every term on the right is a smaller instance of the same question.

7.5.3 The cut-and-paste argument

How do you *prove* a problem has optimal substructure? Almost always by the same three-line argument, which goes by the name **cut and paste**.

> **THE CUT-AND-PASTE PROOF.** Suppose S is an optimal solution to P, decomposing into $S_1, \dots, S_r$ for subproblems $P_1, \dots, P_r$. Suppose, for contradiction, that some S_i is *not* optimal for P_i — that is, there exists a better solution S_i' for P_i. **Cut** S_i out of S and **paste** S_i' in its place. The result is still a feasible solution to P (this is the step you must check), and it has strictly better value than S. That contradicts the optimality of S. Hence every S_i was optimal after all. ■

The step in bold is the one most often skipped, and skipping it is how wrong proofs are born. Pasting a better piece in must leave you with something *legal*. In matrix-chain multiplication it does, because any parenthesization of $A_i \cdots A_k$ can sit inside any parenthesization of the whole. In problems where the pieces interact — where using a cheaper left half forbids something you needed on the right — the paste is illegal, the argument collapses, and dynamic programming does not apply.

7.5.4 A problem *without* optimal substructure

To make the definition bite, here is a problem that does not have the property.

In a directed graph, the **longest simple path** from u to v is the longest path that visits no vertex twice. Does it have optimal substructure? Suppose the longest simple path from u to v passes through w. Is the piece from u to w then the longest simple u-to-w path? Here is a four-vertex graph that says no. Take vertices q, r, s, t arranged in a square $q - r - t - s - q$, with every edge present in *both* directions: $q \to r$ and $r \to q$, $r \to t$ and $t \to r$, $t \to s$ and $s \to t$, $s \to q$ and $q \to s$.

- *The whole problem.* A simple path from q to t can go round either side of the square: $q \to r \to t$ or $q \to s \to t$. Each has length 2, and no simple path is longer, because continuing past t is pointless and r and s are not joined. So $q \to r \to t$ is a longest simple path from q to t.
- *Its first piece.* Inside that answer, the piece from q to r is the single edge $q \to r$, of length 1. But the longest simple path from q to r is $q \to s \to t \to r$, of length 3. The piece is **not** optimal for its own subproblem.
- *The paste.* Could we paste the longer piece in? Take the longest simple path $q \to s \to t \to r$ and follow it with the longest simple path from r to t, which by the same reasoning is $r \to q \to s \to t$. The result, $q \to s \to t \to r \to q \to s \to t$, visits q, s and t twice: it is not a simple path at all.

The two subproblems are not independent — they compete for the same vertices — so the paste is illegal, and optimal pieces do not assemble into an optimal whole.

THE MORAL. Before writing a single line of dynamic-programming code, *prove* optimal substructure with a cut-and-paste argument, and check that the paste produces a legal solution. Shortest paths have optimal substructure; longest simple paths do not, and no amount of table-filling will rescue them. (Longest simple path is in fact NP-hard.)

7.6 Multiplying a Chain of Matrices

This is the central worked problem of the chapter, and it is the best first example of dynamic programming ever devised, because the thing being optimized is not the arithmetic but the *order in which you do the arithmetic*.

7.6.1 The cost of one matrix product

First, the ground rules. A **matrix** is a rectangular grid of numbers. We say a matrix is $n \times m$ ("n by m") if it has n rows and m columns. **Figure 7.8** shows how two matrices are multiplied.

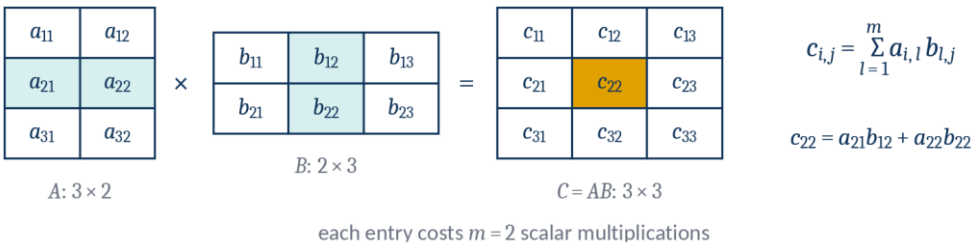


Figure 7.8 Multiplying an $n \times m$ matrix A by an $m \times k$ matrix B, drawn here with $n = 3, m = 2, k = 3$. The shaded row of A and shaded column of B combine to produce the single shaded entry c_{22} of the result, by the formula $c_{i,j} = \sum_{l=1}^{m} a_{i,l} \cdot b_{l,j}$.

Read **Figure 7.8** carefully. To multiply A ($n \times m$) by B ($m \times k$):

- The two matrices must be **conformable**: the number of columns of A must equal the number of rows of B. Both are m here. If they differ, the product does not exist.
- The result C is $n \times k$: as many rows as A, as many columns as B.
- Entry $c_{i,j}$ of the result is obtained from row i of A and column j of B:

$$c_{i,j} = \sum_{l=1}^{m} a_{i,l} \cdot b_{l,j}. \tag{7.7}$$

The symbol $\sum$ (Greek capital sigma) means "add up". Equation Equation (7.7) says: let l run from 1 to m; for each l, multiply the l-th entry of row i of A by the l-th entry of column j of B; add all m of those products together.

Now count the multiplications. Each of the $n \times k$ entries of C costs m scalar multiplications. Therefore:

THE COST RULE. Multiplying an $n \times m$ matrix by an $m \times k$ matrix costs $n \cdot m \cdot k$ scalar multiplications. This one formula is the entire cost model for the rest of §7.6. Memorize it.

A concrete instance: a 10×20 matrix times a 20×3 matrix costs $10 \times 20 \times 3 = 600$ multiplications and produces a 10×3 result.

7.6.2 Associativity, and why the grouping matters

Matrix multiplication is **associative**: $(AB)C = A(BC)$. The two sides contain the same numbers in the same positions. What they do *not* contain is the same amount of work.

To see why not, consider the three matrices

- A: 10×100
- B: 100×5
- C: 5×50

Computing $(AB)C$:

1. AB costs $10 \times 100 \times 5 = 5\,000$, producing a 10×5 matrix.
2. Multiplying that by C costs $10 \times 5 \times 50 = 2\,500$.
3. Total: $5\,000 + 2\,500 = 7\,500$.

Computing $A(BC)$:

1. BC costs $100 \times 5 \times 50 = 25\,000$, producing a 100×50 matrix.
2. Multiplying A by that costs $10 \times 100 \times 50 = 50\,000$.
3. Total: $25\,000 + 50\,000 = 75\,000$.

Same answer. Ten times the work. Nothing was computed wrongly in the second version; it simply built an enormous intermediate matrix that the first version never created.

> **DEFINITION.** A **parenthesization** (or *bracketing*) of a product $A_1 A_2 \cdots A_n$ is a way of inserting brackets so that every multiplication is between exactly two matrices. A product is **fully parenthesized** if it is a single matrix, or a bracketed product of two fully parenthesized products. The **matrix-chain multiplication problem** is: given the dimensions of $A_1, \ldots, A_n$, find a parenthesization minimizing the total number of scalar multiplications.

We are not being asked to multiply anything. We are being asked to produce a *plan*. The output is a set of brackets.

7.6.3 A four-matrix example, priced

Figure 7.9 is a small instance, and it is worth working through every digit.

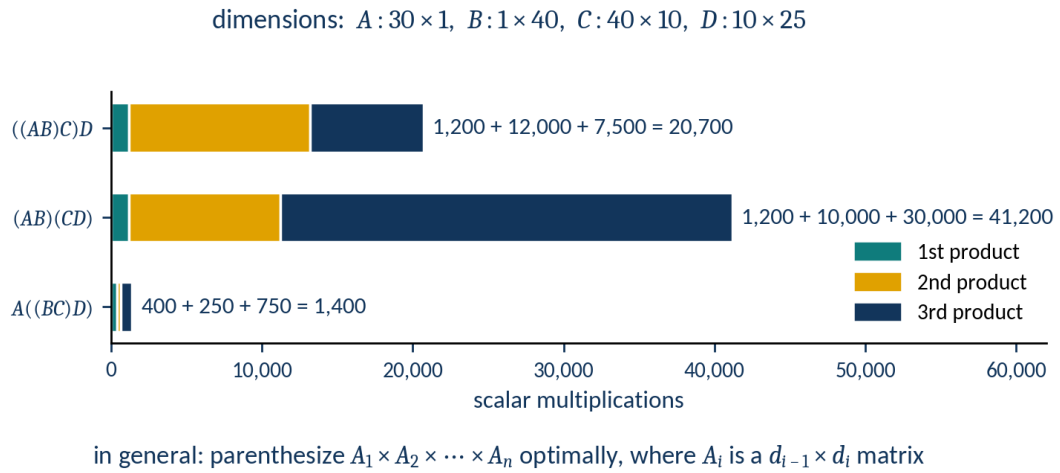


Figure 7.9 A four-matrix example. For A (30×1), B (1×40), C (40×10), D (10×25), three of the five possible bracketings cost 20 700, 41 200 and 1 400 respectively; each bar splits the total into its three products. The last line fixes the notation used from here on: we optimally parenthesize $A_1 \times A_2 \times \cdots \times A_n$, where A_i is a $d_{i-1} \times d_i$ matrix.

The dimension convention on the last line of **Figure 7.9** is the key piece of notation for the whole section, so let us unpack it. Instead of listing each matrix's two dimensions separately — which would repeat every number, since consecutive matrices must be conformable — we list a single sequence

$$d = \langle d_0, d_1, d_2, \dots, d_n \rangle$$

of $n + 1$ numbers, and declare that A_i is $d_{i-1} \times d_i$. Conformability is then automatic: the columns of A_i and the rows of A_{i+1} are both d_i.

> **NOTATION.** This book writes the dimension sequence as d and the cost table as M. Cormen *et al.* [3] write the same sequence as p and the same table as m, so that the recurrence reads $m[i,j] = \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}$. **They are the same thing**, letter for letter: $d \equiv p$ and $M \equiv m$. We use d and M throughout so that the text agrees with the figures, and we quote the p-and-m form once, here, so that you recognize it in CLRS and elsewhere.

For the example of **Figure 7.9**, A is 30×1, B is 1×40, C is 40×10, D is 10×25, so

$$d = \langle 30,\ 1,\ 40,\ 10,\ 25 \rangle, \qquad n = 4.$$

Now price the three bracketings in the figure, one multiplication per line.

Bracketing 1: $((AB)C)D$**.**

1. AB: $(30 \times 1)(1 \times 40)$ costs $30 \times 1 \times 40 = 1\,200$; result 30×40.
2. $(AB)C$: $(30 \times 40)(40 \times 10)$ costs $30 \times 40 \times 10 = 12\,000$; result 30×10.
3. $((AB)C)D$: $(30 \times 10)(10 \times 25)$ costs $30 \times 10 \times 25 = 7\,500$; result 30×25.
4. **Total:** $1\,200 + 12\,000 + 7\,500 = 20\,700$**.**

Bracketing 2: $(AB)(CD)$**.**

1. AB: costs $30 \times 1 \times 40 = 1\,200$; result 30×40.
2. CD: $(40 \times 10)(10 \times 25)$ costs $40 \times 10 \times 25 = 10\,000$; result 40×25.
3. $(AB)(CD)$: $(30 \times 40)(40 \times 25)$ costs $30 \times 40 \times 25 = 30\,000$; result 30×25.
4. **Total:** $1\,200 + 10\,000 + 30\,000 = 41\,200$**.**

Bracketing 3: $A((BC)D)$**.**

1. BC: $(1 \times 40)(40 \times 10)$ costs $1 \times 40 \times 10 = 400$; result 1×10.
2. $(BC)D$: $(1 \times 10)(10 \times 25)$ costs $1 \times 10 \times 25 = 250$; result 1×25.
3. $A((BC)D)$: $(30 \times 1)(1 \times 25)$ costs $30 \times 1 \times 25 = 750$; result 30×25.
4. **Total:** $400 + 250 + 750 = 1\,400$**.**

Every figure matches **Figure 7.9**. And look at the spread: the worst of these three costs 41 200 multiplications, the best costs 1 400. That is a factor of **twenty-nine**, for four small matrices, with no change whatsoever to the answer produced.

> **WHY THE THIRD ONE WINS.** B is 1×40 — one row. Multiplying anything by a one-row matrix is cheap, because the n in $n \cdot m \cdot k$ is 1. Bracketing 3 uses B early and keeps every intermediate matrix one row tall. Bracketings 1 and 2 build fat 30×40 and 40×25 intermediates and then pay to multiply them. The lesson generalizes: a good parenthesization makes the *small* dimensions do the work and never materializes a big intermediate it does not need.

7.6.4 How many parenthesizations are there?

The obvious algorithm is: try them all. Before we accept that, we should count them.

Let $P(n)$ be the number of ways to fully parenthesize a product of n matrices. A parenthesization of $A_1 \cdots A_n$ has an outermost multiplication, which splits the chain between A_k and A_{k+1} for some k with $1 \le k \le n-1$. The left part can be parenthesized in $P(k)$ ways and the right part in $P(n-k)$ ways, independently. So we get the recurrence Equation (7.8):

$$P(1) = 1, \qquad P(n) = \sum_{k=1}^{n-1} P(k)\, P(n-k) \quad (n \ge 2). \tag{7.8}$$

Let us evaluate it by hand for the first few values:

- $P(1) = 1$ — a single matrix, nothing to bracket.
- $P(2) = P(1)P(1) = 1$ — only $(A_1 A_2)$.
- $P(3) = P(1)P(2) + P(2)P(1) = 1 + 1 = 2$ — namely $(A_1(A_2A_3))$ and $((A_1A_2)A_3)$.
- $P(4) = P(1)P(3) + P(2)P(2) + P(3)P(1) = 2 + 1 + 2 = 5$.
- $P(5) = P(1)P(4) + P(2)P(3) + P(3)P(2) + P(4)P(1) = 5 + 2 + 2 + 5 = 14$.

These are the **Catalan numbers**. In closed form, $P(n+1) = C_n$, where C_n is given by Equation (7.9):

$$C_n = \frac{1}{n+1}\binom{2n}{n}, \tag{7.9}$$

and Stanley [4] catalogs over two hundred different families of objects they count. Their growth is known to be

$$C_n = \Omega\left(\frac{4^n}{n^{3/2}}\right),$$

which is exponential in n. So brute-force enumeration is hopeless. **Listing 7.3** enumerates them for a four-matrix chain — five bracketings, all listed and priced — and would take longer than the age of the universe at $n = 60$.

Listing 7.3 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`.

```
// 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;
}
```

On the chain we are about to solve by hand, $d = \langle 10,20,3,5,30 \rangle$, it prints:

```
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, exactly as $P(4) = 5$ predicted, and the best is 4.77 times cheaper than the worst. Keep the number 1 950 and the grouping $((A_1A_2)(A_3A_4))$ in mind: in §7.6.8 we will derive both by hand, without enumerating anything.

7.6.5 The subproblem, and the recurrence derived

Figure 7.10 is the step that turns the problem into a recurrence, and it does so with two observations.

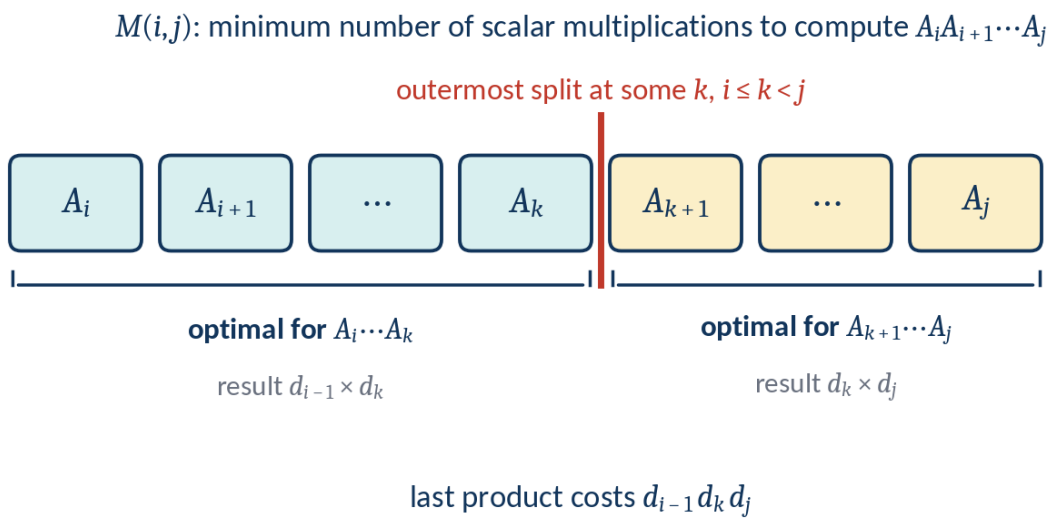


Figure 7.10 The two key observations. First, the outermost bracket must split the chain at *some* position k with $i \le k < j$. Second, whatever k is, the parenthesizations on either side of it must themselves be optimal — optimal substructure, and the justification for the recurrence.

Take them one at a time.

Defining the subproblem. Let us write: *let $M(i,j)$ be the minimum number of multiplications necessary to compute $\prod_{k=i}^{j} A_k$*, that is, the product $A_iA_{i+1}\cdots A_j$. This is the crucial modeling decision of the whole algorithm. Our subproblems are **contiguous subchains**, identified by their first index i and last index j, with $1 \le i \le j \le n$. The original problem is $M(1,n)$.

Why contiguous? Because a parenthesization can only ever bracket *adjacent* matrices; there is no way to multiply A_1 by A_3 without A_2 between them. The structure of the problem hands us the subproblem space.

Observation one: the outermost split. Any full parenthesization of $A_i \cdots A_j$ (with $i < j$) ends with one final multiplication, of two already-computed matrices. Those two are $A_i \cdots A_k$ and $A_{k+1} \cdots A_j$ for some k with $i \le k < j$. There is no other possibility: the last multiplication must combine a prefix with the matching suffix.

What does that final multiplication cost? The left factor $A_i \cdots A_k$ is $d_{i-1} \times d_k$ (rows of A_i, columns of A_k). The right factor $A_{k+1} \cdots A_j$ is $d_k \times d_j$. By the cost rule of §7.6.1, multiplying them costs

$$d_{i-1} \cdot d_k \cdot d_j.$$

Note that this cost depends **only** on i, j and k — not at all on how the two halves were bracketed internally. That independence is what makes the recurrence work.

Observation two: the halves must be optimal. This is optimal substructure, and here is the cut-and-paste proof.

> **CLAIM.** If an optimal parenthesization of $A_i \cdots A_j$ splits at k, then the parenthesization it uses for $A_i \cdots A_k$ is optimal for that subchain, and likewise for $A_{k+1} \cdots A_j$.
>
> **Proof.** Suppose the optimal parenthesization S of $A_i \cdots A_j$ splits at k, and its left part uses a bracketing S_L of cost c_L. Suppose some other bracketing $S_L{}'$ of $A_i \cdots A_k$ costs $c_L{}' < c_L$. **Cut** S_L out of S and **paste** $S_L{}'$ in. The result is still a legal parenthesization of $A_i \cdots A_j$ — a bracketing of the left subchain can be substituted freely, because the dimensions of the left factor, $d_{i-1} \times d_k$, do not depend on how it was bracketed. Its cost is $c_L{}' + c_R + d_{i-1}d_kd_j < c_L + c_R + d_{i-1}d_kd_j$,

the cost of S. That contradicts S being optimal. Hence no such $S_L{}'$ exists, and S_L was optimal. The same argument applies to the right part. ■

Putting the two together. We do not know which k the optimum uses. So we try all of them and keep the best. That is the recurrence, and **Figure 7.11** states it.

$$M(i,i) = 0$$

$$M(i,j) = \min_{i \le k < j} \{ M(i,k) + M(k+1,j) + d_{i-1}\, d_k\, d_j \}$$

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.

In our typography, and with the base case spelled out, it is Equation (7.10):

$$M(i,j) = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{ M(i,k) + M(k+1,j) + d_{i-1}\, d_k\, d_j \} & \text{if } i < j. \end{cases} \tag{7.10}$$

Read the second line in words, because it is the sentence you should be able to produce from memory: *the cheapest way to multiply the subchain from i to j is found by trying every place k where the last multiplication could happen, and for each one adding three things — the cheapest cost of everything to the left of the split, the cheapest cost of everything to the right of the split, and the cost of that last multiplication itself — and then taking the smallest total.*

The base case says a single matrix needs no multiplications at all, which is true by inspection: $M(i,i) = 0$.

THE THREE INDEX TRAPS. (1) The range is $i \le k < j$, **not** $i \le k \le j$; taking $k = j$ would put every matrix on the left and none on the right. (2) The right subchain starts at $k + 1$, not at k; A_k belongs to the left. (3) The dimension triple is d_{i-1}, d_k, d_j — note the $i - 1$, which is why the sequence d has $n + 1$ entries and is indexed from 0. Getting any of these wrong produces a program that runs, prints a number, and is silently wrong.

7.6.6 Counting the subproblems

Why will dynamic programming help here? For exactly the reason it helped with Fibonacci: there are far fewer distinct subproblems than the recursion visits.

A subproblem is a pair (i,j) with $1 \le i \le j \le n$. How many such pairs are there? Those with $i < j$ number $\binom{n}{2}$ — choose two distinct indices from n and let the smaller be i — and those with $i = j$ number n. So the total, Equation (7.11), is

$$\binom{n}{2} + n = \frac{n(n-1)}{2} + n = \frac{n(n+1)}{2} = \Theta(n^2), \tag{7.11}$$

which is precisely the count shown in **Figure 7.12**. For $n = 60$ that is 1 830 distinct questions, against the $P(60) \approx 4.1 \times 10^{32}$ parenthesizations a brute-force search would examine. **Figure 7.12** makes the point that follows from it.

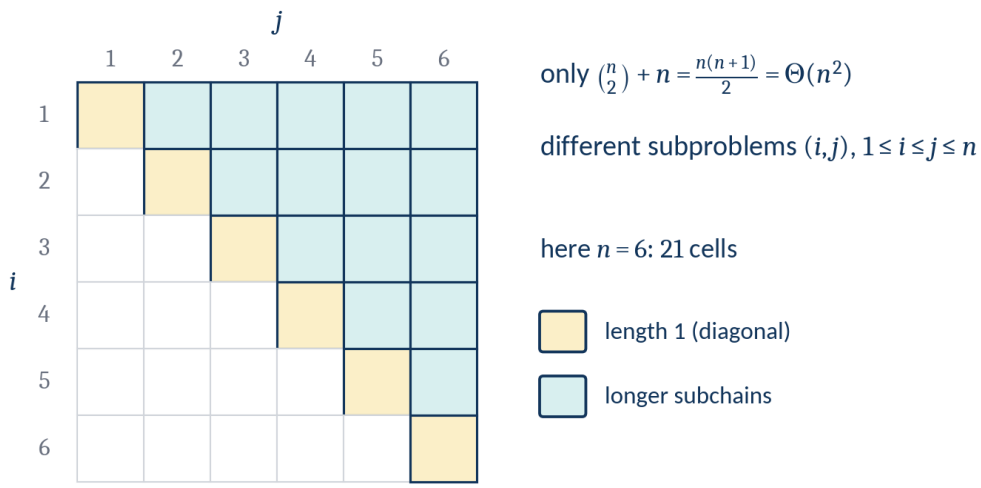


Figure 7.12 The count of the subproblems: there are only $\binom{n}{2} + n = \Theta(n^2)$ different subproblems (i, j) with $1 \le i \le j \le n$. The triangle draws them for $n = 6$: the 21 cells on and above the diagonal. That gap between $\Theta(n^2)$ questions and exponentially many recursive visits is the opening dynamic programming exploits.

7.6.7 The algorithm

The recurrence is not yet an algorithm, because it does not say in what order to evaluate the entries. **Pseudocode 7.5** gives the answer: **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
```

Line 3 is the heart of the matter and deserves a paragraph to itself.

> **WHY LOOP OVER LENGTH.** Entry $M[i, j]$ depends on $M[i, k]$ and $M[k + 1, j]$ for various k strictly between. Both of those are *shorter* subchains: $M[i, k]$ has length $k - i + 1 \le j - i$, and $M[k + 1, j]$ has length $j - k \le j - i$, while $M[i, j]$ has length $j - i + 1$. So if we process subchains in increasing order of length, everything a cell needs is already in the table when we arrive at it. Looping over i or j in the natural order does **not** have this property and produces garbage.

Line by line, what the pseudocode does:

- **Lines 1–2.** For each i, set $M[i, i] \leftarrow 0$. These are the base cases: chains of length 1 are free.
- **Line 3.** `for l ← 2 to n` — consider chains of length 2,3, … , n in that order.

- **Line 4.** `for i ← 1 to n-l+1` — the starting index of a chain of length l can be anywhere from 1 up to $n - l + 1$, beyond which the chain would run off the end.
- **Line 5.** `j ← i+l-1` — the ending index follows from the start and the length.
- **Line 6.** `M[i,j] ← ∞` — initialize the running minimum to "infinity" so that the first candidate always beats it. In C++ we use a very large constant.
- **Lines 7–8.** For each split k from i to $j - 1$, compute the candidate cost $q = M[i, k] + M[k + 1, j] + d_{i-1}d_k d_j$.
- **Lines 9–11.** If q beats the best so far, record both the new best cost in $M[i, j]$ and the winning split point in $c[i, j]$.
- **Line 12.** Return both tables.

The c table is pure bookkeeping — it costs nothing to maintain and it is the only reason we will be able to reconstruct the actual bracketing rather than merely its price.

7.6.8 The table filled by hand, cell by cell

Let us run the algorithm on $d = [10,20,3,5,30]$, with nothing hidden.

The chain is $n = 4$ matrices:

- A_1 is $d_0 \times d_1 = 10 \times 20$
- A_2 is $d_1 \times d_2 = 20 \times 3$
- A_3 is $d_2 \times d_3 = 3 \times 5$
- A_4 is $d_3 \times d_4 = 5 \times 30$

Step 0: the diagonal. $M[1,1] = M[2,2] = M[3,3] = M[4,4] = 0$. A single matrix costs nothing.

Step 1: chains of length $l = 2$. The start index i runs from 1 to $n - l + 1 = 3$. For $l = 2$ there is only one possible split, $k = i$, so no minimization is needed.

Cell $M[1,2]$ ($i = 1, j = 2, k = 1$ only):

$$M[1,1] + M[2,2] + d_0 d_1 d_2 = 0 + 0 + 10 \times 20 \times 3 = 600.$$

So $M[1,2] = 600$ and $c[1,2] = 1$.

Cell $M[2,3]$ ($i = 2, j = 3, k = 2$ only):

$$M[2,2] + M[3,3] + d_1 d_2 d_3 = 0 + 0 + 20 \times 3 \times 5 = 300.$$

So $M[2,3] = 300$ and $c[2,3] = 2$.

Cell $M[3,4]$ ($i = 3, j = 4, k = 3$ only):

$$M[3,3] + M[4,4] + d_2 d_3 d_4 = 0 + 0 + 3 \times 5 \times 30 = 450.$$

So $M[3,4] = 450$ and $c[3,4] = 3$.

Step 2: chains of length $l = 3$. Now i runs from 1 to $n - l + 1 = 2$, and each cell has two candidate splits.

Cell $M[1,3]$ ($i = 1, j = 3$; k may be 1 or 2):

- $k = 1$: $M[1,1] + M[2,3] + d_0 d_1 d_3 = 0 + 300 + 10 \times 20 \times 5 = 0 + 300 + 1\,000 = 1\,300$.
- $k = 2$: $M[1,2] + M[3,3] + d_0 d_2 d_3 = 600 + 0 + 10 \times 3 \times 5 = 600 + 0 + 150 = 750$.

- The minimum is 750, attained at $k = 2$. So $M[1,3] = 750$, $c[1,3] = 2$.

Pause on that comparison. Splitting at $k = 1$ means computing A_2A_3 first (cheap, 300) and then multiplying the tall A_1 by the result — but the last multiplication is $10 \times 20 \times 5 = 1\,000$, which dominates. Splitting at $k = 2$ means paying 600 for A_1A_2 up front, but that collapses the 20 down to a 3, so the last multiplication costs only 150. Paying more early to shrink a dimension is the recurring theme of this problem.

Cell $M[2,4]$ ($i = 2, j = 4$; k may be 2 or 3):

- $k = 2$: $M[2,2] + M[3,4] + d_1d_2d_4 = 0 + 450 + 20 \times 3 \times 30 = 0 + 450 + 1\,800 = 2\,250$.
- $k = 3$: $M[2,3] + M[4,4] + d_1d_3d_4 = 300 + 0 + 20 \times 5 \times 30 = 300 + 0 + 3\,000 = 3\,300$.
- The minimum is 2 250, attained at $k = 2$. So $M[2,4] = 2\,250$, $c[2,4] = 2$.

Step 3: the whole chain, $l = 4$**.** Only $i = 1$, hence $j = 4$, with three candidate splits.

- $k = 1$: $M[1,1] + M[2,4] + d_0d_1d_4 = 0 + 2\,250 + 10 \times 20 \times 30 = 0 + 2\,250 + 6\,000 = 8\,250$.
- $k = 2$: $M[1,2] + M[3,4] + d_0d_2d_4 = 600 + 450 + 10 \times 3 \times 30 = 600 + 450 + 900 = 1\,950$.
- $k = 3$: $M[1,3] + M[4,4] + d_0d_3d_4 = 750 + 0 + 10 \times 5 \times 30 = 750 + 0 + 1\,500 = 2\,250$.
- The minimum is 1 950, attained at $k = 2$. So $M[1,4] = 1\,950$, $c[1,4] = 2$.

That is the answer: 1 950 **scalar multiplications**, exactly the number **Listing 7.3** found by enumerating all five bracketings. The finished tables are **Table 7.3** and **Table 7.4**.

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
$i = 4$	—	—	—	—

Notice the order in which the cells were filled: the diagonal first, then the diagonal just above it, then the next, ending at the single corner cell. The table is filled **diagonal by diagonal**, moving up and to the right — which is exactly what looping over l produces.

7.6.9 The c table and the reconstruction

Figure 7.13 states what the two tables give us.

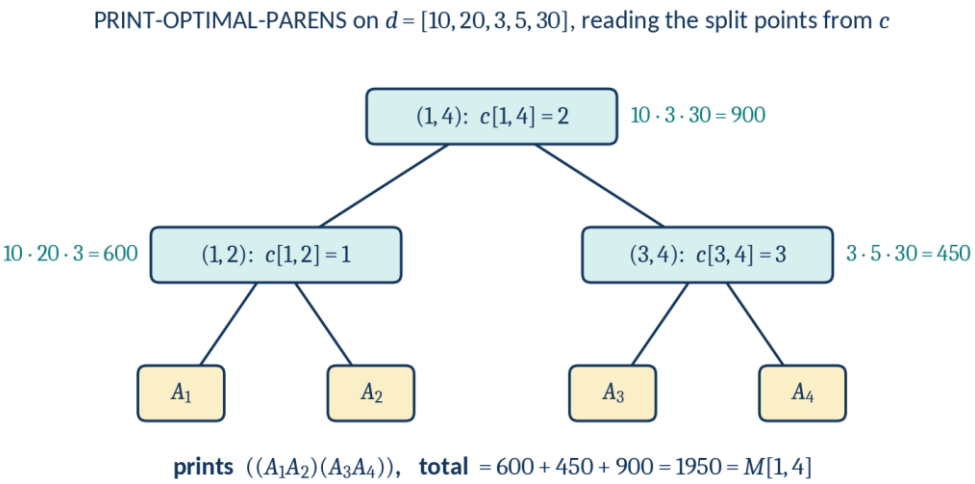


Figure 7.13 After the algorithm runs, $M[1,n]$ holds the optimal value and c holds the optimal subdivisions; a simple recursive procedure **Print-Optimal-Parens**(c, i, j) walks c and prints the bracketing. The tree shows that walk on the example we have just solved by hand, $d = [10,20,3,5,30]$, with the cost of each product beside its call.

Print-Optimal-Parens is three lines, shown in **Pseudocode 7.6**:

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) + ")";
}
```

Let us run it by hand on **Table 7.4**, one call per line.

1. `parens(1,4)`: $i \neq j$, and $c[1,4] = 2$. So the answer is `"(" + parens(1,2) + parens(3,4) + ")"`.
2. `parens(1,2)`: $i \neq j$, and $c[1,2] = 1$. So it is `"(" + parens(1,1) + parens(2,2) + ")"`.
3. `parens(1,1)`: $i = j$, returns `"A1"`.
4. `parens(2,2)`: $i = j$, returns `"A2"`.
5. Back in step 2: `"(A1A2)"`.
6. `parens(3,4)`: $i \neq j$, and $c[3,4] = 3$. So it is `"(" + parens(3,3) + parens(4,4) + ")"`.
7. `parens(3,3)` returns `"A3"`; `parens(4,4)` returns `"A4"`.
8. Back in step 6: `"(A3A4)"`.
9. Back in step 1: `"((A1A2)(A3A4))"`.

The optimal bracketing is $((A_1A_2)(A_3A_4))$. Let us verify its price independently, from the cost rule alone:

1. A_1A_2: $(10 \times 20)(20 \times 3)$ costs $10 \times 20 \times 3 = 600$; result 10×3.
2. A_3A_4: $(3 \times 5)(5 \times 30)$ costs $3 \times 5 \times 30 = 450$; result 3×30.
3. The final product: $(10 \times 3)(3 \times 30)$ costs $10 \times 3 \times 30 = 900$; result 10×30.
4. Total: $600 + 450 + 900 = 1\,950$. ✓

It agrees with $M[1,4]$ and with the brute-force enumeration. Three independent routes to the same number.

WHY YOU MUST KEEP c. The M table alone tells you the optimum *costs* 1 950; it does not tell you how to achieve it. You could recover the split points by re-examining M (for each k, check whether $M[i,k] + M[k+1,j] + d_{i-1}d_kd_j$ equals $M[i,j]$), at a cost of $\Theta(n)$ per reconstruction step. Recording c as you go costs $\Theta(n^2)$ extra memory and makes reconstruction $\Theta(n)$ overall. This is the general pattern of **step 4** of the DP recipe in §7.8: store the choice, not just the value.

7.6.10 $\Theta(n^3)$ time and $\Theta(n^2)$ space

Figure 7.14 gives the verdict on the cost.

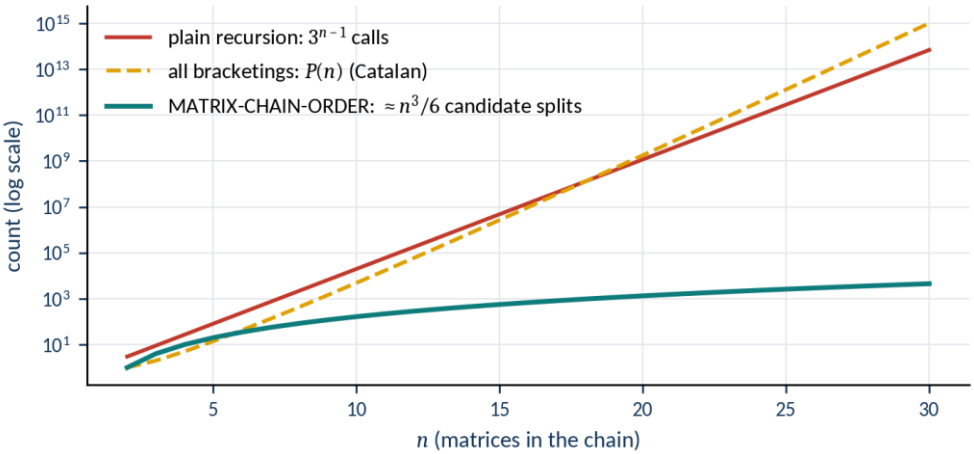


Figure 7.14 The running time of **Matrix-Chain-Order**: easily seen to be $O(n^3)$, and in fact also $\Omega(n^3)$, hence $\Theta(n^3)$. The plot, on a logarithmic scale, sets its candidate-split count against the 3^{n-1} calls of plain recursion and the $P(n)$ bracketings of brute force: *from exponential time to polynomial*, the whole point of the chapter in five words.

Let us prove both bounds rather than assert them.

The upper bound. The algorithm has three nested loops. The outer loop over l runs at most n times. The middle loop over i runs at most n times. The inner loop over k runs at most n times. Each innermost iteration does $O(1)$ work — two table lookups, two multiplications, two additions, a comparison and possibly two stores. Hence $O(n \cdot n \cdot n \cdot 1) = O(n^3)$.

The lower bound. Count exactly. For a given l, the middle loop runs $n - l + 1$ times, and for each of those the inner loop runs $j - i = l - 1$ times. So the exact number of innermost iterations is

$$\sum_{l=2}^{n} (n - l + 1)\,(l - 1).$$

Substitute $t = l - 1$, so t runs from 1 to $n - 1$ and $n - l + 1 = n - t$:

$$\sum_{t=1}^{n-1} (n - t)\ t \ = \ n \sum_{t=1}^{n-1} t \ - \ \sum_{t=1}^{n-1} t^2 \ = \ n \cdot \frac{(n-1)n}{2} \ - \ \frac{(n-1)n(2n-1)}{6}.$$

Putting both terms over a denominator of 6 gives Equation (7.12):

$$= \frac{3n^2(n-1) - (n-1)n(2n-1)}{6} = \frac{n(n-1)[3n-(2n-1)]}{6}$$
$$= \frac{n(n-1)(n+1)}{6} = \frac{n^3-n}{6}. \tag{7.12}$$

So the innermost statement executes exactly $(n^3 - n)/6$ times — a clean formula worth remembering. For $n = 4$ that is $(64 - 4)/6 = 10$ iterations, and indeed the trace in §7.6.8 evaluated exactly ten candidate splits: $1 + 1 + 1$ for $l = 2$, $2 + 2$ for $l = 3$, and 3 for $l = 4$. Since $(n^3 - n)/6 = \Theta(n^3)$, the running time is $\Theta(n^3)$, not merely $O(n^3)$.

The space. Two tables of size $n \times n$, so $\Theta(n^2)$ words. As **Pseudocode 7.5** makes clear, only the upper triangle is ever used — about $n^2/2$ cells each — so a careful implementation can halve the constant, but the order of growth stays $\Theta(n^2)$.

> **THE HEADLINE.** Brute force examines $P(n) = \Omega(4^n/n^{3/2})$ parenthesizations. Dynamic programming examines $(n^3 - n)/6$ candidate splits. At $n = 25$ that is 1 289 904 147 324 against 2 600 — a factor of roughly half a billion. This is what "from exponential time to polynomial" buys.

7.6.11 The complete program

Listing 7.4 implements **Matrix-Chain-Order** and **Print-Optimal-Parens** exactly as **Pseudocode 7.5** gives them, printing every candidate split as it is evaluated so that the machine's trace can be compared line by line with the hand trace of §7.6.8.

Listing 7.4 **Matrix-Chain-Order** and **Print-Optimal-Parens** in C++, with a full trace of every candidate split. Compile with `g++ -std=c++17 -O2 -o mcm mcm.cpp` and run `./mcm`.

```
// mcm.cpp - Matrix-Chain-Order exactly as Section 7.6.7 states it, with a full trace.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <climits>
using namespace std;

int n;                        // number of matrices
vector<long long> d;          // d[0..n]: A_i is d[i-1] x d[i]
vector<vector<long long>> M; // M[i][j] = cheapest cost for A_i..A_j
vector<vector<int>> c;        // c[i][j] = the k that achieves it

void matrixChainOrder(bool trace) {
    M.assign(n + 1, vector<long long>(n + 1, 0));
    c.assign(n + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= n; ++i) M[i][i] = 0;             // one matrix: nothing to
multiply
    for (int l = 2; l <= n; ++l)                          // l = chain length
        for (int i = 1; i <= n - l + 1; ++i) {
            int j = i + l - 1;
            M[i][j] = LLONG_MAX;
            for (int k = i; k <= j - 1; ++k) {
                long long q = M[i][k] + M[k+1][j] + d[i-1]*d[k]*d[j];
                if (trace)
                    cout << "  l=" << l << "  M[" << i << "," << j << "]  k=" << k
                         << ":  M[" << i << "," << k << "]=" << M[i][k]
                         << " + M[" << k+1 << "," << j << "]=" << M[k+1][j]
                         << " + " << d[i-1] << "*" << d[k] << "*" << d[j]
                         << "=" << d[i-1]*d[k]*d[j] << "  ->  " << q << "\n";
                if (q < M[i][j]) { M[i][j] = q; c[i][j] = k; }
            }
            if (trace) cout << "  => M[" << i << "," << j << "] = " << M[i][j]
                            << ", c[" << i << "," << j << "] = " << c[i][j] << "\n\n";
        }
}

string printOptimalParens(int i, int j) {                 // Print-Optimal-Parens of
Section 7.6.9
    if (i == j) return "A" + to_string(i);
    return "(" + printOptimalParens(i, c[i][j])
               + printOptimalParens(c[i][j] + 1, j) + ")";
}
```

Listing 7.4 (Continued) *Matrix-Chain-Order and Print-Optimal-Parens in C++, with a full trace of every candidate split. Compile with g++ -std=c++17 -O2 -o mcm mcm.cpp and run ./mcm.*

```
void showTables() {
    cout << "  M table (blank below the diagonal):\n      ";
    for (int j = 1; j <= n; ++j) cout << setw(8) << ("j=" + to_string(j));
    cout << "\n";
    for (int i = 1; i <= n; ++i) {
        cout << "  i=" << i << " ";
        for (int j = 1; j <= n; ++j) cout << setw(8) << (j < i ? string(".") :
to_string(M[i][j]));
        cout << "\n";
    }
    cout << "\n  c table (only j > i is meaningful):\n      ";
    for (int j = 1; j <= n; ++j) cout << setw(8) << ("j=" + to_string(j));
    cout << "\n";
    for (int i = 1; i <= n; ++i) {
        cout << "  i=" << i << " ";
        for (int j = 1; j <= n; ++j) cout << setw(8) << (j <= i ? string("-") :
to_string(c[i][j]));
        cout << "\n";
    }
}

int main() {
    // ---- The example of Section 7.6.8: d = [10, 20, 3, 5, 30] ----
    d = {10, 20, 3, 5, 30};
    n = (int)d.size() - 1;
    cout << "CHAIN 1: d = [10, 20, 3, 5, 30]  (A1 10x20, A2 20x3, A3 3x5, A4 5x30)\n\n";
    matrixChainOrder(true);
    showTables();
    cout << "\n  optimal cost      = " << M[1][n] << "\n";
    cout << "  optimal grouping  = " << printOptimalParens(1, n) << "\n";

    // ---- The earlier example of Section 7.6.3 ----
    d = {30, 1, 40, 10, 25};
    n = (int)d.size() - 1;
    cout << "\n\nCHAIN 2: d = [30, 1, 40, 10, 25]  (A 30x1, B 1x40, C 40x10, D 10x25)\n";
    matrixChainOrder(false);
    showTables();
    cout << "\n  optimal cost      = " << M[1][n] << "\n";
    cout << "  optimal grouping  = " << printOptimalParens(1, n) << "\n";
    return 0;
}
```

The output, in full:

```
CHAIN 1: d = [10, 20, 3, 5, 30]  (A1 10x20, A2 20x3, A3 3x5, A4 5x30)

  l=2  M[1,2]  k=1:  M[1,1]=0 + M[2,2]=0 + 10*20*3=600  ->  600
  => M[1,2] = 600, c[1,2] = 1

  l=2  M[2,3]  k=2:  M[2,2]=0 + M[3,3]=0 + 20*3*5=300  ->  300
  => M[2,3] = 300, c[2,3] = 2

  l=2  M[3,4]  k=3:  M[3,3]=0 + M[4,4]=0 + 3*5*30=450  ->  450
  => M[3,4] = 450, c[3,4] = 3

  l=3  M[1,3]  k=1:  M[1,1]=0 + M[2,3]=300 + 10*20*5=1000  ->  1300
  l=3  M[1,3]  k=2:  M[1,2]=600 + M[3,3]=0 + 10*3*5=150  ->  750
  => M[1,3] = 750, c[1,3] = 2

  l=3  M[2,4]  k=2:  M[2,2]=0 + M[3,4]=450 + 20*3*30=1800  ->  2250
  l=3  M[2,4]  k=3:  M[2,3]=300 + M[4,4]=0 + 20*5*30=3000  ->  3300
  => M[2,4] = 2250, c[2,4] = 2

  l=4  M[1,4]  k=1:  M[1,1]=0 + M[2,4]=2250 + 10*20*30=6000  ->  8250
  l=4  M[1,4]  k=2:  M[1,2]=600 + M[3,4]=450 + 10*3*30=900  ->  1950
  l=4  M[1,4]  k=3:  M[1,3]=750 + M[4,4]=0 + 10*5*30=1500  ->  2250
  => M[1,4] = 1950, c[1,4] = 2

  M table (blank below the diagonal):
           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

  c table (only j > i is meaningful):
           j=1     j=2     j=3     j=4
  i=1        -       1       2       2
  i=2        -       -       2       2
  i=3        -       -       -       3
  i=4        -       -       -       -

  optimal cost      = 1950
  optimal grouping  = ((A1A2)(A3A4))


CHAIN 2: d = [30, 1, 40, 10, 25]  (A 30x1, B 1x40, C 40x10, D 10x25)
  M table (blank below the diagonal):
           j=1     j=2     j=3     j=4
  i=1        0    1200     700    1400
  i=2        .       0     400     650
  i=3        .       .       0   10000
  i=4        .       .       .       0

  c table (only j > i is meaningful):
           j=1     j=2     j=3     j=4
  i=1        -       1       1       1
  i=2        -       -       2       3
  i=3        -       -       -       3
  i=4        -       -       -       -

  optimal cost      = 1400
  optimal grouping  = (A1((A2A3)A4))
```

Compare the machine's ten trace lines with the ten candidate splits computed by hand in §7.6.8: 600; 300; 450; 1 300 and 750; 2 250 and 3 300; 8 250, 1 950 and 2 250. Every one matches. Both tables match **Table 7.3** and **Table 7.4** cell for cell, and the reconstruction matches the hand walk of §7.6.9.

Chain 2 is a bonus: it is the example from **Figure 7.9**, and the program confirms that the third bracketing priced by hand in §7.6.3, $A((BC)D)$ at 1 400, is not merely cheaper than the other two shown — it is the global optimum over all five.

7.7 Memoization: Keeping the Recursion

7.7.1 Lookup-Chain

Bottom-up filling is efficient, but it has a cost of its own: you must work out the correct order of evaluation and encode it in the loops. For matrix chains that order was "by increasing length", which took a paragraph of justification. For a more complicated recurrence it can be genuinely hard to see.

Memoization sidesteps the problem entirely. The recursion discovers the right order by itself — it simply asks for what it needs, and whatever it asks for gets computed before the answer comes back. **Pseudocode 7.7** gives this version.

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]
```

Line 1, repeated as **Pseudocode 7.8**, is the entire difference between this and an exponential disaster:

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]
```

If the entry has been written, return it. Otherwise fall through, compute it the slow way, store it, and return. The sentinel value ∞ serves two purposes at once: it means "not computed yet" and it also loses every comparison, so no special-casing is needed when the minimum is first taken.

> **A SENTINEL MUST BE IMPOSSIBLE.** Using ∞ as "not yet computed" works here only because no real cost is ever ∞. If you choose a sentinel that a genuine answer could equal — say 0, when a genuine answer of 0 is possible — the memo table will silently recompute those entries for ever, and your "memoized" algorithm will still be exponential. A separate boolean `computed[i][j]` array is always safe and costs one bit per cell.

The running time of **Lookup-Chain** is the same $\Theta(n^3)$, by the accounting trick of §7.4.1: there are $\Theta(n^2)$ table entries, filling one entry costs $O(n)$ (the loop over k, with the recursive calls charged to their own entries), so the total is $\Theta(n^2) \cdot O(n) = O(n^3)$.

7.7.2 Measured: plain recursion, memoized, bottom-up

Listing 7.5 puts all three on the same footing — the same recurrence, the same random chains, the same machine — and also computes the Catalan counts of §7.6.4.

Listing 7.5 The same matrix-chain recurrence implemented three ways — plain recursion, memoized recursion, bottom-up table — checked against each other and timed. Compile with `g++ -std=c++17 -O2 -o mcm_compare mcm_compare.cpp`.

```
// mcm_compare.cpp - three implementations of the same recurrence, timed side by side,
// plus a count of how many parenthesizations there are (the Catalan numbers).
#include <iostream>
#include <iomanip>
#include <vector>
#include <chrono>
#include <climits>
#include <random>
using namespace std;
using namespace std::chrono;

int n;
vector<long long> d;
long long calls;

// (a) The recurrence translated straight into recursion. No table.
long long plain(int i, int j) {
    ++calls;
    if (i == j) return 0;
    long long best = LLONG_MAX;
    for (int k = i; k < j; ++k)
        best = min(best, plain(i, k) + plain(k + 1, j) + d[i-1]*d[k]*d[j]);
    return best;
}

// (b) The same recursion with a table of remembered answers: Lookup-Chain.
vector<vector<long long>> Mm;
long long lookupChain(int i, int j) {
    ++calls;
    if (Mm[i][j] != LLONG_MAX) return Mm[i][j];   // solved before: return the stored value
    if (i == j) return Mm[i][j] = 0;
    long long best = LLONG_MAX;
    for (int k = i; k < j; ++k)
        best = min(best, lookupChain(i,k) + lookupChain(k+1,j) + d[i-1]*d[k]*d[j]);
    return Mm[i][j] = best;
}
```

Listing 7.5 (Continued) *The same matrix-chain recurrence implemented three ways — plain recursion, memoized recursion, bottom-up table — checked against each other and timed. Compile with g++ -std=c++17 -O2 -o mcm_compare mcm_compare.cpp.*

```
// (c) Bottom-up: Matrix-Chain-Order.
long long bottomUp() {
    vector<vector<long long>> M(n+2, vector<long long>(n+2, 0));
    for (int l = 2; l <= n; ++l)
        for (int i = 1; i <= n - l + 1; ++i) {
            int j = i + l - 1; M[i][j] = LLONG_MAX;
            for (int k = i; k < j; ++k)
                M[i][j] = min(M[i][j], M[i][k] + M[k+1][j] + d[i-1]*d[k]*d[j]);
        }
    return M[1][n];
}

int main() {
    // How many ways can a chain of n matrices be parenthesized? P(1)=1,
    // P(n) = sum over k of P(k)P(n-k): the Catalan numbers.
    cout << "n :  parenthesizations P(n)\n";
    vector<double> P(26, 0); P[1] = 1;
    for (int m = 2; m <= 25; ++m) for (int k = 1; k < m; ++k) P[m] += P[k]*P[m-k];
    for (int m : {1,2,3,4,5,6,10,15,20,25})
        cout << setw(3) << m << " :  " << fixed << setprecision(0) << P[m] << "\n";

    cout << "\nSame answer, three implementations, on random chains:\n\n";
    cout << left << setw(5) << "n" << setw(12) << "answer"
         << setw(14) << "plain calls" << setw(12) << "plain ms"
         << setw(14) << "memo calls" << setw(12) << "memo ms"
         << setw(12) << "bottom ms" << "\n";
    mt19937 rng(12345);
    for (int sz : {4, 8, 12, 15, 17, 60, 200}) {
        n = sz;
        d.assign(n + 1, 0);
        for (int i = 0; i <= n; ++i) d[i] = 1 + rng() % 50;

        long long ansB; double tb;
        { auto t0 = steady_clock::now(); ansB = bottomUp();
          tb = duration<double,milli>(steady_clock::now()-t0).count(); }

        long long ansM; double tm; long long memoCalls;
        { Mm.assign(n+2, vector<long long>(n+2, LLONG_MAX)); calls = 0;
          auto t0 = steady_clock::now(); ansM = lookupChain(1, n);
          tm = duration<double,milli>(steady_clock::now()-t0).count(); memoCalls = calls; }
```

Listing 7.5 (Continued) *The same matrix-chain recurrence implemented three ways — plain recursion, memoized recursion, bottom-up table — checked against each other and timed. Compile with g++ -std=c++17 -O2 -o mcm_compare mcm_compare.cpp.*

```
        cout << left << setw(5) << n << setw(12) << ansB;
        if (n <= 17) {                              // beyond this the plain version is
hopeless
            calls = 0;
            auto t0 = steady_clock::now(); long long ansP = plain(1, n);
            double tp = duration<double,milli>(steady_clock::now()-t0).count();
            if (ansP != ansB) { cout << "MISMATCH\n"; return 1; }
            cout << setw(14) << calls << setw(12) << fixed << setprecision(3) << tp;
        } else cout << setw(14) << "(not run)" << setw(12) << "-";
        if (ansM != ansB) { cout << "MISMATCH\n"; return 1; }
        cout << setw(14) << memoCalls << setw(12) << fixed << setprecision(3) << tm
             << setw(12) << tb << "\n";
    }
    return 0;
}
```

Its output:

```
n :  parenthesizations P(n)
  1 :  1
  2 :  1
  3 :  2
  4 :  5
  5 :  14
  6 :  42
 10 :  4862
 15 :  2674440
 20 :  1767263190
 25 :  1289904147324

Same answer, three implementations, on random chains:

n    answer      plain calls   plain ms    memo calls    memo ms     bottom ms
4    15596       27            0.000       21            0.000       0.003
8    3839        2187          0.006       169           0.001       0.001
12   16508       177147        0.501       573           0.002       0.003
15   62862       4782969       12.033      1121          0.004       0.004
17   14523       43046721      109.037     1633          0.005       0.006
60   44364       (not run)     -           71981         0.208       0.076
200  143061      (not run)     -           2666601       6.834       1.601
```

The first table confirms §7.6.4 exactly: $P(4) = 5$, $P(5) = 14$, $P(25) = 1\,289\,904\,147\,324$.

The second table is worth studying column by column.

The plain-recursion call counts are exactly 3^{n-1}. Check: $3^3 = 27$ for $n = 4$; $3^7 = 2\,187$ for $n = 8$; $3^{11} = 177\,147$ for $n = 12$; $3^{16} = 43\,046\,721$ for $n = 17$. That is not a coincidence. Write $C(L)$ for the calls made on a subchain of length L; the call recurrence $C(i,j) = 1 + \sum_{k=i}^{j-1}(C(i,k) + C(k+1,j))$ becomes $C(L) = 1 + 2\sum_{t=1}^{L-1} C(t)$, which gives $C(1) = 1$, $C(2) = 3$, $C(3) = 9$ and, by induction, $C(L) = 3^{L-1}$. For every size in this table that is even more than the number of parenthesizations — 43 million calls against $P(17) = 35\,357\,670$ bracketings — because the plain recursion re-derives shared subchains once per enclosing split. (Beyond about $n = 20$ the Catalan numbers, which grow like 4^n, overtake 3^{n-1}; both are hopeless.)

The memoized call counts are polynomial. 1 633 calls at $n = 17$ against 43 million. At $n = 200$, memoization makes 2 666 601 calls — and $2\,666\,601 = (200^3 - 200)/3 + 1$, which is exactly what the accounting predicts: each of the $(n^3 - n)/6$ triples (i,j,k) makes two calls, one for each half, and the outermost call adds one.

The timings: at $n = 17$, plain takes 109.037 ms, memoized takes 0.005 ms. Memoization is about twenty-two thousand times faster, from one extra line of code.

And the trade-off, which is the point of this section. At $n = 60$ memoized takes 0.208 ms while bottom-up takes 0.076 ms; at $n = 200$ the gap is 6.834 ms against 1.601 ms. Bottom-up is consistently about four times faster. Both are $\Theta(n^3)$. The difference is entirely constant factors: function-call overhead, stack frames, and a memory access pattern that jumps around the table instead of sweeping it.

7.7.3 The trade-off, stated

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

WHICH TO USE. Use **memoization** when the recurrence is awkward to order, when only a small fraction of the subproblems will actually be needed, or when you are writing the first version and want it to be obviously correct. Use **bottom-up** when every subproblem will be solved anyway, when the constant factor matters, when the recursion would be too deep for the stack, or when you want to exploit a space saving of the kind in §7.4.3 and §7.9.10. When in doubt, write the memoized version first, check it, then convert it to bottom-up and check the two agree — which is exactly what **Listing 7.5** does.

7.8 The Four Steps of Designing a Dynamic-Programming Algorithm

We have now done the same thing twice, for Fibonacci and for matrix chains. It is time to extract the general method, in four steps, because it is a recipe you can apply to a problem you have never seen. Kleinberg and Tardos [5, ch. 6] give a complementary presentation of the same recipe, built around a different set of examples.

7.8.1 Step 1: characterize the structure of an optimal solution

Figure 7.15 is the first step.

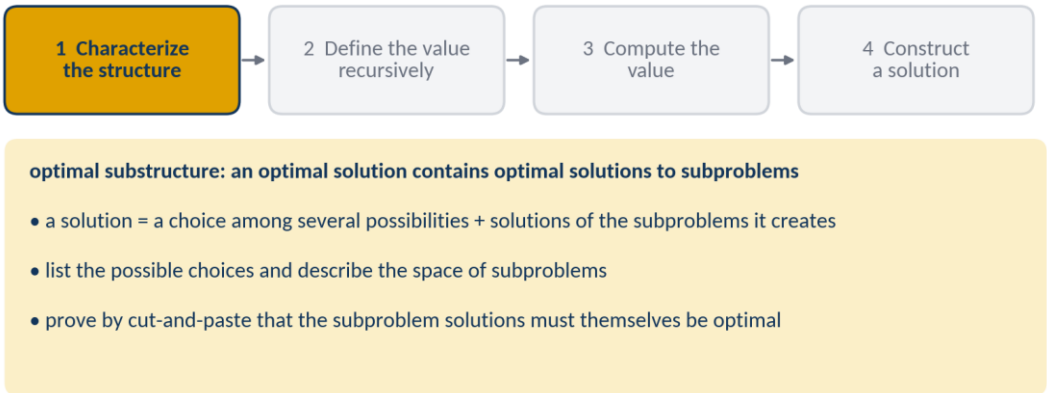


Figure 7.15 Step 1 of the recipe: show optimal substructure. A solution is a choice out of a number of possibilities plus solutions to the subproblems that choice creates; then show, by cut-and-paste, that those subproblem solutions must themselves be optimal.

The step breaks into three questions you must answer in order.

What are the possible choices? At the top level, what decision does the algorithm make? For matrix chains, the choice is *where to place the outermost bracket* — the split point k. For the longest common subsequence in §7.9, it will be *what to do with the last character of each string*. Naming the choice is the creative act; everything after it is mechanical.

What subproblems does a choice create? Having made the choice, what smaller instances of the same problem remain? This defines the **space of subproblems**, and you must characterize it precisely, because the table you will build has one cell per subproblem. Matrix chains: contiguous subchains (i, j), so $\Theta(n^2)$ cells. LCS: prefix pairs (i, j), so $\Theta(mn)$ cells.

Are the subsolutions necessarily optimal? Prove it, by the cut-and-paste argument of §7.5.3. Do not skip this. If the answer is no, dynamic programming does not apply and you must find a different method.

7.8.2 Step 2: recursively define the value of an optimal solution

Figure 7.16 is the second step.

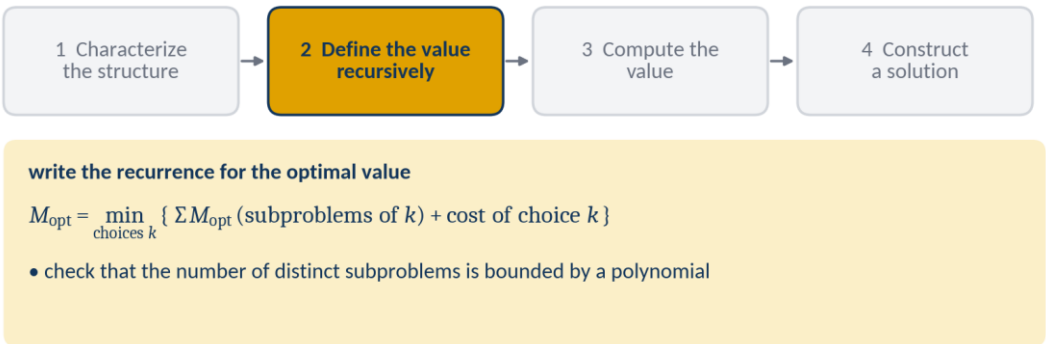


Figure 7.16 Step 2: write the recurrence. Its general form is $M_{opt} = \min_{\text{choices } k}\{(\text{sum of } M_{opt} \text{ of the subproblems produced by } k) + (\text{cost of making choice } k)\}$ — and then show the number of distinct subproblem instances is bounded by a polynomial.

The template in the figure is worth writing out in full, because every dynamic program in this book is an instance of it:

$$M_{\text{opt}}(\text{problem}) = \min_{\text{all choices } k} \left\{ \sum_{\text{subproblems } s \text{ created by } k} M_{\text{opt}}(s) + \text{cost}(k) \right\}. \tag{7.13}$$

Check the matrix-chain recurrence Equation (7.10) against Equation (7.13): the choices are the split points k; the subproblems created by choice k are (i, k) and $(k + 1, j)$; the cost of making that choice is $d_{i-1} d_k d_j$. It fits exactly.

The second half of **Figure 7.16** is the sanity check that must not be skipped: **show that the number of different subproblem instances is bounded by a polynomial.** If it is not — if your subproblems are, say, all 2^n subsets of the input — then the table has exponentially many cells and filling it is no better than brute force. A polynomial number of subproblems is precisely the condition called **overlapping subproblems** (§7.10), and together with optimal substructure it is one of the two hallmarks of a dynamic-programming problem.

7.8.3 Steps 3 and 4: compute the value, then construct the solution

Figure 7.17 gives the last two steps together.

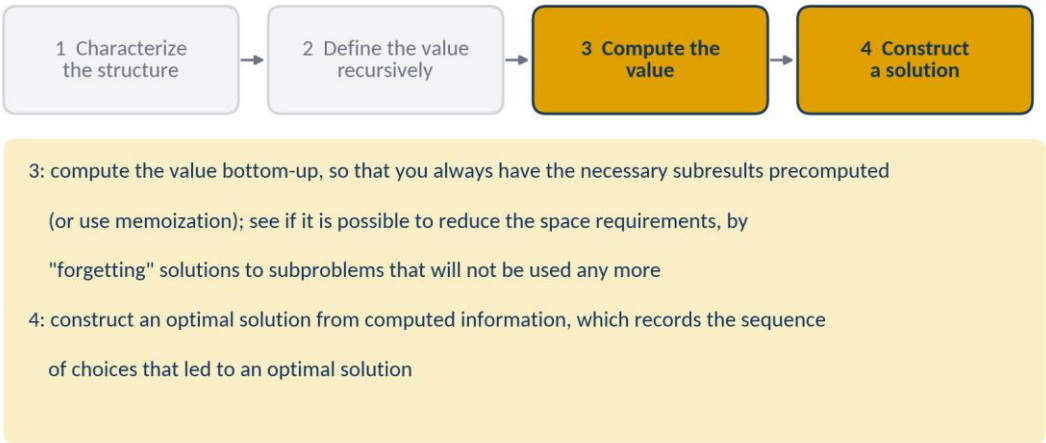


Figure 7.17 Steps 3 and 4: compute the optimal value bottom-up so the needed subresults are always ready (or use memoization), look for a chance to reduce the space by forgetting subproblems that will not be used again, and finally construct an optimal solution from the recorded sequence of choices.

Step 3: compute the value of an optimal solution, bottom-up. The phrase to remember is *so that you always have the necessary subresults precomputed*. That is the entire requirement on your loop order: when you compute a cell, everything it depends on must already be done. If choosing that order is painful, the parenthetical *(or use memoization)* is your escape hatch.

Figure 7.17 then adds the space optimization: *see if it is possible to reduce the space requirements, by "forgetting" solutions to subproblems that will not be used any more.* We did this for Fibonacci in §7.4.3, going from $\Theta(n)$ to $\Theta(1)$, and we shall do it for LCS in §7.9.10, going from $\Theta(mn)$ to $\Theta(\min(m,n))$.

Step 4: construct an optimal solution from computed information. The key words are *which records the sequence of choices that led to an optimal solution*. This is the c table of §7.6.9 and the b table of §7.9.8. Step 4 is optional — omit it if you only need the value — and omitting it is often what makes the space saving of step 3 possible, because a reconstruction table cannot be forgotten as you go.

7.8.4 The recipe, on one page

THE FOUR STEPS. 1. Characterize the structure of an optimal solution. Name the choice; name the subproblems that choice creates; prove by cut-and-paste that the subsolutions must be optimal. **2. Recursively define** the value of an optimal solution: $M_{\text{opt}} = \min_k\{\sum M_{\text{opt}}(\text{subproblems}) + \text{cost}(k)\}$, with base cases. Check the subproblem count is polynomial. **3. Compute** the value, bottom-up in an order that respects the dependencies (or top-down with memoization). Consider forgetting what you no longer need. **4. Construct** an optimal solution from a table of recorded choices — if you need the solution and not just its value.

THE MOST COMMON FAILURE. Beginners jump straight to step 3 — they start drawing a table before they have named the subproblem. The table is the *last* thing to think about, not the first. If you cannot finish the sentence "the subproblem is: given …, find the best …", you are not ready to write any code.

7.9 Longest Common Subsequence

7.9.1 Why anyone cares

Two pieces of text are given, and we want a number saying how similar they are. Three uses stand out: comparing DNA sequences in the study of how species evolved; spell checking; and, generally, any question of the form "how much do these two strings have in common?". Version-control systems answer `diff` with exactly this computation; plagiarism detectors, fuzzy search boxes and the Profitina Laundry case study of §7.11 all rest on it.

One of the standard measures of similarity is the length of the **longest common subsequence**, and it is a textbook dynamic program — the second of this chapter's two big worked examples.

7.9.2 Subsequence, defined

Figure 7.18 gives the definition.

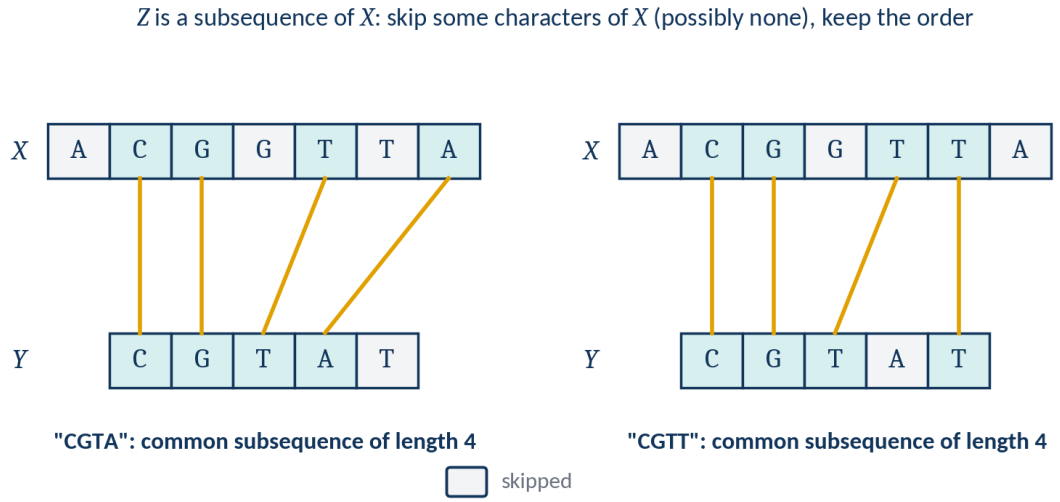


Figure 7.18 The definition: Z is a subsequence of X if Z can be generated by skipping some (possibly none) characters of X. For $X = $ "ACGGTTA" and $Y = $ "CGTAT", LCS(X, Y) is "CGTA" or "CGTT". Solving the problem means finding the *skips*.

DEFINITION. Let $X = x_1 x_2 \cdots x_m$ be a string. A string $Z = z_1 z_2 \cdots z_r$ is a **subsequence** of X if there are indices $i_1 < i_2 < \cdots < i_r$ with $z_t = x_{i_t}$ for every t. In words: Z is obtained from X by crossing out some (possibly no) characters and keeping the rest **in their original order**.

SUBSEQUENCE IS NOT SUBSTRING. A **substring** must be contiguous; a **subsequence** need not be. In "ACGGTTA", the string "CGG" is both a substring and a subsequence, while "AGT" is a subsequence (positions 1, 3, 5) but not a substring. Confusing the two is the single most common error in this topic.

DEFINITION. Z is a **common subsequence** of X and Y if it is a subsequence of both. A **longest common subsequence** of X and Y, written LCS(X, Y), is a common subsequence of greatest length. Note "a", not "the": there may be several, and they may all have the same length.

Work the example of **Figure 7.18** by hand. With $X = $ "ACGGTTA" and $Y = $ "CGTAT":

- Is "CGT" common? In X it is positions 2, 3, 5 (C, G, T) — increasing ✓. In Y it is positions 1, 2, 3 — increasing ✓. Yes, length 3.
- Is "CGTA" common? In X: positions 2, 3, 5, 7 ✓. In Y: positions 1, 2, 3, 4 ✓. Yes, length 4.
- Is "CGTT" common? In X: positions 2, 3, 5, 6 ✓. In Y: positions 1, 2, 3, 5 ✓. Yes, length 4.

- Is "CGTAT" common? It is all of Y. In X we would need C, G, T, A, T at increasing positions: C at 2, G at 3, T at 5, A at 7, then another T after position 7 — there is none. No.

So the longest common subsequences have length 4, and both "CGTA" and "CGTT" achieve it, exactly as **Figure 7.18** says.

7.9.3 Brute force, and why it fails

The obvious algorithm: list every subsequence of X, test each against Y, keep the longest that passes.

How many subsequences does a string of length m have? Each of the m characters is either kept or crossed out, independently, so there are 2^m choices (some producing duplicate strings, but at most 2^m). Checking whether a given candidate of length up to m is a subsequence of Y takes $O(n)$ by a single scan. Total: $O(n\,2^m)$.

For two strings of length 20, that is about a million checks — survivable. For two DNA sequences of length 1000, it is 2^{1000}, a number with 302 digits. Brute force is not an option, and we need the dynamic program.

7.9.4 Optimal substructure, proved

Figure 7.19 gives the derivation, and it is the whole algorithm in one picture.

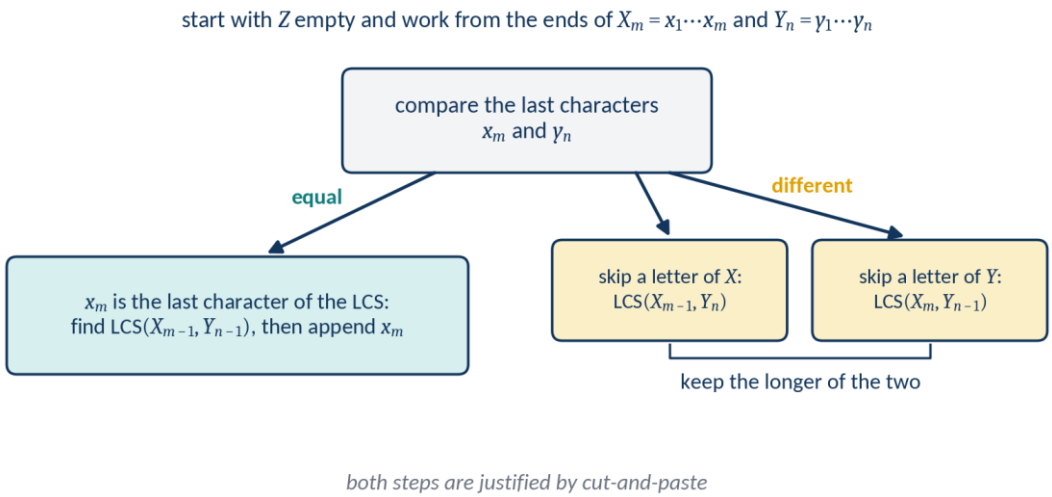


Figure 7.19 The optimal substructure of LCS. Working from the *ends* of X_m and Y_n: if the last characters agree, that character joins the answer and we recurse on both strings shortened by one; if they disagree, we must skip a letter from X or a letter from Y, and we decide which by comparing the two resulting LCS lengths. The justification is the cut-and-paste argument.

First, notation. Write X_i for the **prefix** $x_1 x_2 \cdots x_i$ of X — the first i characters — with X_0 the empty string. Likewise Y_j. The subproblems will be all pairs of prefixes.

Now the two cases, and their proofs.

THEOREM (optimal substructure of LCS). Let $X = x_1 \cdots x_m$ and $Y = y_1 \cdots y_n$, and let $Z = z_1 \cdots z_r$ be *any* longest common subsequence of X and Y.

(1) If $x_m = y_n$, then $z_r = x_m = y_n$, and Z_{r-1} is an LCS of X_{m-1} and Y_{n-1}. **(2)** If $x_m \neq y_n$ and $z_r \neq x_m$, then Z is an LCS of X_{m-1} and Y_n. **(3)** If $x_m \neq y_n$ and $z_r \neq y_n$, then Z is an LCS of X_m and Y_{n-1}.

Proof of (1). Suppose $x_m = y_n = ch$.

First, $z_r = ch$. Suppose not. Then append ch to Z, giving a string of length $r + 1$. Is it a common subsequence? In X, the characters of Z occupy positions all at most m, and since $z_r \neq x_m$ they are all at most $m - 1$; so appending x_m at position m keeps the indices increasing ✓. The same argument works in Y ✓. So we have a common subsequence of length $r + 1$, contradicting that Z was longest. Hence $z_r = ch$.

Second, Z_{r-1} *is an LCS of* X_{m-1} *and* Y_{n-1}. It is certainly a common subsequence of them: delete the last character from Z and from both strings. Suppose it were not *longest* — suppose some common subsequence W of X_{m-1} and Y_{n-1} had length greater than $r - 1$. **Cut** Z_{r-1} out and **paste** W in: W followed by ch is a common subsequence of X and Y (legal, because W's indices are all $< m$ in X and $< n$ in Y) of length greater than r. Contradiction. ■

Proof of (2). Suppose $x_m \neq y_n$ and $z_r \neq x_m$. Then no character of Z uses position m of X (only the last one could, since indices increase), so Z is a common subsequence of X_{m-1} and Y. If some longer common subsequence W of X_{m-1} and Y existed, then W would also be a common subsequence of X and Y — every subsequence of a prefix is a subsequence of the whole — and longer than Z. Contradiction. ■

Proof of (3) is (2) with the roles of X and Y exchanged. ■

WHY THE THEOREM IS COMPLETE. If $x_m \neq y_n$, then z_r cannot equal both x_m and y_n, so at least one of cases (2) and (3) applies. Together with case (1) that covers every possibility, so the recurrence below is exhaustive. When both (2) and (3) apply we may take either, which is why ties can be broken arbitrarily and why there can be several LCSs.

7.9.5 The recurrence

Following step 2 of the recipe, define the table:

DEFINITION. Let $c[i, j]$ be the length of a longest common subsequence of the prefixes X_i and Y_j. The answer to the original problem is $c[m, n]$.

Figure 7.20 gives the 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.

Written out, it is Equation (7.14):

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max\{c[i, j-1],\ c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases} \tag{7.14}$$

Each line, in words:

- **Line 1 (base case).** If either string is empty, the only common subsequence is the empty string, of length 0.
- **Line 2 (match).** If the last characters agree, theorem part (1) says that character is in the answer, and the rest of the answer is the LCS of the two shortened prefixes. Hence one more than the *diagonal* neighbor.
- **Line 3 (mismatch).** If they disagree, theorem parts (2) and (3) say the answer is the better of two possibilities: drop the last character of X (giving the neighbor *above*, $c[i-1, j]$), or drop the last character of Y (giving the neighbor to the *left*, $c[i, j-1]$). Take the larger.

> **TWO INDEXING CONVENTIONS AT ONCE.** The table is indexed from 0 ($c[0, j]$ and $c[i, 0]$ are the empty-prefix rows) while the strings are indexed from 1 (x_1 is the first character). In C++ a `string` is indexed from 0, so the character compared at table position (i, j) is `X[i-1]` against `Y[j-1]`. Every off-by-one bug in LCS code lives in that line.

How many subproblems? Pairs (i, j) with $0 \le i \le m$ and $0 \le j \le n$: exactly $(m+1)(n+1) = \Theta(mn)$. Polynomial ✓ — step 2's sanity check passes.

7.9.6 The algorithm

Pseudocode 7.9 gives **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
```

The second table b is step 4 of the recipe. At each cell it records the *choice* that produced the value:

- `"copy"` — the characters matched; the value came from the diagonal.
- `"skipx"` — they did not match and the value came from above, $c[i-1, j]$; we skipped a character of X.
- `"skipy"` — they did not match and the value came from the left, $c[i, j-1]$; we skipped a character of Y.

We draw these three as arrows — ↖ for copy, ↑ for skip-x, ← for skip-y — because reconstruction is literally following arrows backward.

Note line 10 of the pseudocode: `else if c[i-1,j] ≥ c[i,j-1]`. The comparison is $\geq$, not $>$, so ties go to `skipx`. Any consistent rule is correct; this one is the rule of **Pseudocode 7.9**, and we follow it so that the table below is reproducible.

The loop order here is the simple one: row by row, left to right. It works because $c[i, j]$ depends only on $c[i-1, j-1]$, $c[i-1, j]$ and $c[i, j-1]$ — all in the previous row, or earlier in this row.

7.9.7 The table filled by hand

Figure 7.21 shows where we are going: the finished tables for $X = $ "ACGGTTA" and $Y = $ "CGTAT". Here is the run that fills them, one cell per line.

		0	1 C	2 G	3 T	4 A	5 T
0		0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1
2	C	0	↖ 1	← 1	← 1	↑ 1	↑ 1
3	G	0	↑ 1	↖ 2	← 2	← 2	← 2
4	G	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 2
5	T	0	↑ 1	↑ 2	↖ 3	← 3	↖ 3
6	T	0	↑ 1	↑ 2	↖ 3	↑ 3	↖ 4
7	A	0	↑ 1	↑ 2	↑ 3	↖ 4	↑ 4

c[7,5] = 4

traceback spells "CGTT"

length only? keep just two rows: row $i-1$ and row i

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?

So $X = x_1x_2x_3x_4x_5x_6x_7 = $ A, C, G, G, T, T, A with $m = 7$, and $Y = y_1y_2y_3y_4y_5 = $ C, G, T, A, T with $n = 5$.

Row 0 and column 0 are all zeros, by the base case.

Row $i = 1$ ($x_1 = $ A).

- $c[1,1]$: $y_1 = $ C, and A $\neq$ C. Compare above $c[0,1] = 0$ with left $c[1,0] = 0$. Tie, so $\geq$ takes above: $c[1,1] = 0$, $b = \uparrow$.
- $c[1,2]$: $y_2 = $ G, A $\neq$ G. Above $c[0,2] = 0$, left $c[1,1] = 0$. Tie: $c[1,2] = 0$, $b = \uparrow$.
- $c[1,3]$: $y_3 = $ T, A $\neq$ T. Above 0, left 0. Tie: $c[1,3] = 0$, $b = \uparrow$.
- $c[1,4]$: $y_4 = $ A, and A $=$ A — a match. Diagonal $c[0,3] = 0$, so $c[1,4] = 0 + 1 = 1$, $b = \nwarrow$.
- $c[1,5]$: $y_5 = $ T, A $\neq$ T. Above $c[0,5] = 0$, left $c[1,4] = 1$. Since $0 \geq 1$ is false, take the left: $c[1,5] = 1$, $b = \leftarrow$.

Row $i = 2$ ($x_2 = $ C).

- $c[2,1]$: $y_1 = $ C — match. Diagonal $c[1,0] = 0$, so $c[2,1] = 1$, $b = \nwarrow$.
- $c[2,2]$: G, no match. Above $c[1,2] = 0$, left $c[2,1] = 1$. $0 \geq 1$ false: $c[2,2] = 1$, $b = \leftarrow$.
- $c[2,3]$: T, no match. Above $c[1,3] = 0$, left $c[2,2] = 1$. $c[2,3] = 1$, $b = \leftarrow$.

- $c[2,4]$: A, no match (C $\neq$ A). Above $c[1,4] = 1$, left $c[2,3] = 1$. $1 \geq 1$ true: $c[2,4] = 1$, $b = \uparrow$.
- $c[2,5]$: T, no match. Above $c[1,5] = 1$, left $c[2,4] = 1$. $1 \geq 1$: $c[2,5] = 1$, $b = \uparrow$.

Row $i = 3$ ($x_3 = $ G).

- $c[3,1]$: C, no match. Above $c[2,1] = 1$, left $c[3,0] = 0$. $1 \geq 0$: $c[3,1] = 1$, $b = \uparrow$.
- $c[3,2]$: G — match. Diagonal $c[2,1] = 1$, so $c[3,2] = 2$, $b = \nwarrow$.
- $c[3,3]$: T, no match. Above $c[2,3] = 1$, left $c[3,2] = 2$. $1 \geq 2$ false: $c[3,3] = 2$, $b = \leftarrow$.
- $c[3,4]$: A, no match. Above $c[2,4] = 1$, left $c[3,3] = 2$: $c[3,4] = 2$, $b = \leftarrow$.
- $c[3,5]$: T, no match. Above $c[2,5] = 1$, left $c[3,4] = 2$: $c[3,5] = 2$, $b = \leftarrow$.

Row $i = 4$ ($x_4 = $ G).

- $c[4,1]$: C, no match. Above $c[3,1] = 1$, left 0. $c[4,1] = 1$, $b = \uparrow$.
- $c[4,2]$: G — match. Diagonal $c[3,1] = 1$: $c[4,2] = 2$, $b = \nwarrow$.
- $c[4,3]$: T, no match. Above $c[3,3] = 2$, left $c[4,2] = 2$. $2 \geq 2$: $c[4,3] = 2$, $b = \uparrow$.
- $c[4,4]$: A, no match. Above $c[3,4] = 2$, left $c[4,3] = 2$: $c[4,4] = 2$, $b = \uparrow$.
- $c[4,5]$: T, no match. Above $c[3,5] = 2$, left $c[4,4] = 2$: $c[4,5] = 2$, $b = \uparrow$.

Row $i = 5$ ($x_5 = $ T).

- $c[5,1]$: C, no match. Above $c[4,1] = 1$, left 0: $c[5,1] = 1$, $b = \uparrow$.
- $c[5,2]$: G, no match. Above $c[4,2] = 2$, left $c[5,1] = 1$. $2 \geq 1$: $c[5,2] = 2$, $b = \uparrow$.
- $c[5,3]$: T — match. Diagonal $c[4,2] = 2$: $c[5,3] = 3$, $b = \nwarrow$.
- $c[5,4]$: A, no match. Above $c[4,4] = 2$, left $c[5,3] = 3$. $2 \geq 3$ false: $c[5,4] = 3$, $b = \leftarrow$.
- $c[5,5]$: T — match. Diagonal $c[4,4] = 2$: $c[5,5] = 3$, $b = \nwarrow$.

Row $i = 6$ ($x_6 = $ T).

- $c[6,1]$: C, no match. Above $c[5,1] = 1$, left 0: $c[6,1] = 1$, $b = \uparrow$.
- $c[6,2]$: G, no match. Above $c[5,2] = 2$, left $c[6,1] = 1$: $c[6,2] = 2$, $b = \uparrow$.
- $c[6,3]$: T — match. Diagonal $c[5,2] = 2$: $c[6,3] = 3$, $b = \nwarrow$.
- $c[6,4]$: A, no match. Above $c[5,4] = 3$, left $c[6,3] = 3$. $3 \geq 3$: $c[6,4] = 3$, $b = \uparrow$.
- $c[6,5]$: T — match. Diagonal $c[5,4] = 3$: $c[6,5] = 4$, $b = \nwarrow$.

Row $i = 7$ ($x_7 = $ A).

- $c[7,1]$: C, no match. Above $c[6,1] = 1$, left 0: $c[7,1] = 1$, $b = \uparrow$.
- $c[7,2]$: G, no match. Above $c[6,2] = 2$, left $c[7,1] = 1$: $c[7,2] = 2$, $b = \uparrow$.
- $c[7,3]$: T, no match. Above $c[6,3] = 3$, left $c[7,2] = 2$. $3 \geq 2$: $c[7,3] = 3$, $b = \uparrow$.
- $c[7,4]$: A — match. Diagonal $c[6,3] = 3$: $c[7,4] = 4$, $b = \nwarrow$.
- $c[7,5]$: T, no match. Above $c[6,5] = 4$, left $c[7,4] = 4$. $4 \geq 4$: $c[7,5] = 4$, $b = \uparrow$.

The finished table is **Table 7.6**, and the answer sits in the bottom-right corner: $c[7,5] = 4$.

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**

Two sanity checks you can apply to any LCS table. First, entries never decrease as you move right or down — adding a character can never shorten the best common subsequence. Second, two horizontally or vertically adjacent entries differ by at most 1. Scan **Table 7.6**: both hold everywhere.

7.9.8 The arrow table

Table 7.7 records, for each cell, which case of the recurrence was used.

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)**	↑	↑	↑	↖	↑

7.9.9 Reconstructing the LCS

Start at (7,5) and follow arrows until an index hits 0. Every ↖ you pass through contributes a character.

1. (7,5): arrow ↑. No character. Move to (6,5).
2. (6,5): arrow ↖. **Emit $x_6 = $ T.** Move to (5,4).

3. (5,4): arrow ←. No character. Move to (5,3).
4. (5,3): arrow ↖. **Emit $x_5 = $ T.** Move to (4,2).
5. (4,2): arrow ↖. **Emit $x_4 = $ G.** Move to (3,1).
6. (3,1): arrow ↑. No character. Move to (2,1).
7. (2,1): arrow ↖. **Emit $x_2 = $ C.** Move to (1,0).
8. (1,0): $j = 0$. Stop.

The characters were emitted in the order T, T, G, C — that is, **backward**, because we walked from the end of the strings toward the beginning. Reversing gives

$$\mathrm{LCS}(X, Y) = \text{"CGTT"}, \qquad \text{of length 4,}$$

which is one of the two answers named in **Figure 7.18**. Verify it directly: "CGTT" sits at positions 2, 3, 5, 6 of "ACGGTTA" ✓ and at positions 1, 2, 3, 5 of "CGTAT" ✓.

> **WHY WE GET "CGTT" AND NOT "CGTA".** Both have length 4 and both are correct answers. Which one the algorithm produces depends entirely on the tie-breaking rule of line 10 of **Pseudocode 7.9**: with ≥, ties go to `skipx`. Change it to $>$ and the walk takes different turns and can produce the other LCS. An algorithm for "a longest common subsequence" is under no obligation to prefer one maximal answer over another.

The reconstruction visits one cell per step and each step decreases $i + j$ by at least 1, so it takes $O(m + n)$ time — negligible beside the $\Theta(mn)$ spent filling the table.

7.9.10 $\Theta(mn)$, and the $\Theta(\min(m, n))$-space variant

Time. Two nested loops, m and n iterations, $O(1)$ work per cell: $\Theta(mn)$. There is no minimization loop as there was in matrix chains, which is why LCS is quadratic and matrix chains cubic.

Space, as written. Two tables of $(m + 1)(n + 1)$ cells: $\Theta(mn)$.

Now the question **Figure 7.21** asks: *how much can we reduce our space requirements, if we do not need to reconstruct LCS?*

Apply the space economy of §7.4.3. Look at the recurrence: $c[i, j]$ depends on $c[i - 1, j - 1]$, $c[i - 1, j]$ and $c[i, j - 1]$. Every one of those is either in row $i - 1$ or earlier in row i. Nothing in rows $0..i - 2$ is ever consulted again once row $i - 1$ is complete. So we need only **two rows**: the previous one and the current one. That is $2(n + 1)$ cells, or $\Theta(n)$.

One more turn of the screw. Because $\mathrm{LCS}(X, Y) = \mathrm{LCS}(Y, X)$, we are free to put whichever string we like along the rows. Putting the **longer** string down the rows and the **shorter** across the columns makes each row $\min(m, n) + 1$ cells wide. Hence:

> **THE SPACE RESULT.** The *length* of an LCS can be computed in $\Theta(mn)$ time and $\Theta(\min(m, n))$ space. The *subsequence itself* cannot be recovered this way, because the b table — the record of choices — has been thrown away. (It can still be recovered in linear space by Hirschberg's divide-and-conquer refinement, at the cost of doubling the running time; see [3, §14.4] and Exercise 7.14.)

That is the answer to the question of **Figure 7.21**. For two DNA strings of a million characters each, the full table would need about 4×10^{12} bytes — four terabytes — while the two-row version needs about eight megabytes.

7.9.11 The complete program

Listing 7.6 implements **LCS-Length** exactly as **Pseudocode 7.9** gives it, prints both tables, reconstructs the subsequence, and then times the full-table version against the two-row version on random strings.

Listing 7.6 **LCS-Length**, the arrow table, the reconstruction, and the two-row variant, with a timing and memory comparison. Compile with `g++ -std=c++17 -O2 -o lcs lcs.cpp`.

```
// lcs.cpp - LCS-Length of Section 7.9.6, with the c table, the b table of arrows,
// the reconstruction, and the two-row space-saving variant.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
#include <chrono>
#include <random>
using namespace std;
using namespace std::chrono;

enum Arrow { NONE, COPY, SKIPX, SKIPY };              // diagonal, up, left
const char* glyph[] = {" ", "\\", "^", "<"};          // \ = copy, ^ = skip x, < = skip y

vector<vector<int>>   c;
vector<vector<Arrow>> b;

void lcsLength(const string& X, const string& Y) {
    int m = (int)X.size(), n = (int)Y.size();
    c.assign(m + 1, vector<int>(n + 1, 0));
    b.assign(m + 1, vector<Arrow>(n + 1, NONE));
    for (int i = 1; i <= m; ++i)
        for (int j = 1; j <= n; ++j) {
            if (X[i-1] == Y[j-1]) {                    // x_i = y_j : the letter joins the
LCS
                c[i][j] = c[i-1][j-1] + 1; b[i][j] = COPY;
            } else if (c[i-1][j] >= c[i][j-1]) {       // tie broken toward skipping x, as
in Section 7.9.6
                c[i][j] = c[i-1][j];       b[i][j] = SKIPX;
            } else {
                c[i][j] = c[i][j-1];       b[i][j] = SKIPY;
            }
        }
}

void reconstruct(const string& X, int i, int j, string& out) {
    if (i == 0 || j == 0) return;
    if (b[i][j] == COPY) { reconstruct(X, i-1, j-1, out); out += X[i-1]; }
    else if (b[i][j] == SKIPX) reconstruct(X, i-1, j, out);
    else                       reconstruct(X, i, j-1, out);
}
```

Listing 7.6 (Continued) *LCS-Length, the arrow table, the reconstruction, and the two-row variant, with a timing and memory comparison. Compile with g++ -std=c++17 -O2 -o lcs lcs.cpp.*

```
// Theta(min(m,n)) space: two rows only, and no b table, so no reconstruction.
int lcsTwoRows(const string& A, const string& B) {
    const string& X = (A.size() >= B.size()) ? A : B;  // keep the SHORT string across the
row
    const string& Y = (A.size() >= B.size()) ? B : A;
    int m = (int)X.size(), n = (int)Y.size();
    vector<int> prev(n + 1, 0), cur(n + 1, 0);
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j)
            cur[j] = (X[i-1] == Y[j-1]) ? prev[j-1] + 1 : max(prev[j], cur[j-1]);
        swap(prev, cur);
    }
    return prev[n];
}

void show(const string& X, const string& Y) {
    int m = (int)X.size(), n = (int)Y.size();
    cout << "  c table:\n        j: ";
    for (int j = 0; j <= n; ++j) cout << setw(4) << j;
    cout << "\n          y: ";
    for (int j = 0; j <= n; ++j) cout << setw(4) << (j == 0 ? '-' : Y[j-1]);
    cout << "\n";
    for (int i = 0; i <= m; ++i) {
        cout << "  i=" << i << " x=" << (i == 0 ? '-' : X[i-1]) << "  ";
        for (int j = 0; j <= n; ++j) cout << setw(4) << c[i][j];
        cout << "\n";
    }
    cout << "\n  b table  ( \\ = copy, ^ = skip x, < = skip y ):\n          j: ";
    for (int j = 1; j <= n; ++j) cout << setw(4) << Y[j-1];
    cout << "\n";
    for (int i = 1; i <= m; ++i) {
        cout << "  i=" << i << " x=" << X[i-1] << "    ";
        for (int j = 1; j <= n; ++j) cout << setw(4) << glyph[b[i][j]];
        cout << "\n";
    }
}
```

Listing 7.6 (Continued) *LCS-Length, the arrow table, the reconstruction, and the two-row variant, with a timing and memory comparison. Compile with g++ -std=c++17 -O2 -o lcs lcs.cpp.*

```
int main() {
    string X = "ACGGTTA", Y = "CGTAT";
    cout << "X = \"" << X << "\"  (m=" << X.size() << ")   Y = \""
         << Y << "\"  (n=" << Y.size() << ")\n\n";
    lcsLength(X, Y);
    show(X, Y);
    string z; reconstruct(X, (int)X.size(), (int)Y.size(), z);
    cout << "\n  LCS length          = " << c[X.size()][Y.size()] << "\n";
    cout << "  one LCS             = \"" << z << "\"\n";
    cout << "  two-row variant     = " << lcsTwoRows(X, Y) << " (same length, no
string)\n";

    // Timing: full table vs two rows, on random DNA-like strings.
    cout << "\n" << left << setw(10) << "m=n" << setw(14) << "LCS len"
         << setw(14) << "full ms" << setw(16) << "full MiB"
         << setw(14) << "2-row ms" << setw(14) << "2-row MiB" << "\n";
    mt19937 rng(7);
    for (int sz : {200, 1000, 4000, 8000}) {
        string A(sz, ' '), B(sz, ' ');
        for (char& ch : A) ch = "ACGT"[rng() % 4];
        for (char& ch : B) ch = "ACGT"[rng() % 4];
        auto t0 = steady_clock::now(); lcsLength(A, B);
        double tf = duration<double,milli>(steady_clock::now()-t0).count();
        int lenF = c[sz][sz];
        auto t1 = steady_clock::now(); int lenT = lcsTwoRows(A, B);
        double tt = duration<double,milli>(steady_clock::now()-t1).count();
        if (lenF != lenT) { cout << "MISMATCH\n"; return 1; }
        double fullMiB  = (double)(sz+1)*(sz+1)*(sizeof(int)+sizeof(Arrow)) / 1048576.0;
        double tworMiB  = (double)2*(sz+1)*sizeof(int) / 1048576.0;
        cout << left << setw(10) << sz << setw(14) << lenF
             << setw(14) << fixed << setprecision(2) << tf
             << setw(16) << setprecision(3) << fullMiB
             << setw(14) << setprecision(2) << tt
             << setw(14) << setprecision(3) << tworMiB << "\n";
        c.clear(); c.shrink_to_fit(); b.clear(); b.shrink_to_fit();
    }
    return 0;
}
```

Its output:

```
X = "ACGGTTA"  (m=7)   Y = "CGTAT"  (n=5)

  c table:
        j:    0   1   2   3   4   5
          y:    -   C   G   T   A   T
  i=0 x=-     0   0   0   0   0   0
  i=1 x=A     0   0   0   0   1   1
  i=2 x=C     0   1   1   1   1   1
  i=3 x=G     0   1   2   2   2   2
  i=4 x=G     0   1   2   2   2   2
  i=5 x=T     0   1   2   3   3   3
  i=6 x=T     0   1   2   3   3   4
  i=7 x=A     0   1   2   3   4   4

  b table  ( \ = copy, ^ = skip x, < = skip y ):
          j:    C   G   T   A   T
  i=1 x=A       ^   ^   ^   \   <
  i=2 x=C       \   <   <   ^   ^
  i=3 x=G       ^   \   <   <   <
  i=4 x=G       ^   \   ^   ^   ^
  i=5 x=T       ^   ^   \   <   \
  i=6 x=T       ^   ^   \   ^   \
  i=7 x=A       ^   ^   ^   \   ^

  LCS length          = 4
  one LCS             = "CGTT"
  two-row variant     = 4 (same length, no string)

m=n       LCS len       full ms       full MiB        2-row ms      2-row MiB
200       124           0.28          0.308           0.03          0.002
1000      645           6.36          7.645           0.74          0.008
4000      2598          247.23        122.131         25.07         0.031
8000      5216          1930.83       488.403         152.86        0.061
```

The machine's c table is **Table 7.6** exactly, cell for cell, and its b table is **Table 7.7** exactly, arrow for arrow. The reconstruction produces "CGTT", as the hand walk of §7.9.9 did.

The timing block makes the space argument concrete. At $m = n = 8000$ the full table needs 488 MiB and takes 1 931 ms; the two-row version needs 0.061 MiB — about eight thousand times less — and takes 153 ms, twelve times *faster*, because it fits in cache and never allocates half a gigabyte. Asymptotically both are $\Theta(mn)$ in time; in practice the one that respects the memory hierarchy wins decisively.

Note also the quadratic growth, and its limits. Doubling both lengths from 4000 to 8000 multiplies mn by 4, so $\Theta(mn)$ predicts a factor of about 4; the two-row version comes close ($152.86/25.07 \approx 6.1$ in this run), while the full table's time grows by $1930.83/247.23 \approx 7.8$, nearly twice the prediction. The excess is memory pressure: at half a gigabyte, the full table no longer fits in any cache, and every row pays for trips to main memory.

7.10 Overlapping Subproblems versus Independent Subproblems

We can now draw the line the chapter has been walking toward.

7.10.1 The two hallmarks

> **DEFINITION.** A problem has **overlapping subproblems** if a recursive algorithm for it solves the *same* subproblem repeatedly, rather than always generating new ones. Equivalently: the number of **distinct** subproblems is small — typically polynomial in the input size — while the recursion tree that visits them is large.

Dynamic programming applies when **both** of these hold:

1. **Optimal substructure** — an optimal solution contains optimal solutions to subproblems (§7.5.2). This is what lets you write the recurrence at all.
2. **Overlapping subproblems** — there are few distinct subproblems and they recur. This is what makes the table worth building.

Divide and conquer, by contrast, needs optimal substructure in a trivial sense but specifically wants the **opposite** of the second hallmark: it wants the subproblems to be *independent and all different*, as the caption of **Figure 7.2** says. If Merge Sort's subproblems overlapped, memoizing it would help; they do not, so a memo table would be pure overhead — every lookup a miss, every entry written once and never read.

> **BOTH HALLMARKS ARE NEEDED.** Optimal substructure without overlap gives divide and conquer (Merge Sort, binary search): build a table and you waste memory for nothing. Overlap without optimal substructure gives a problem where the table is fillable but its entries do not compose into an optimal whole — longest simple path (§7.5.4) is the standard example, and it is NP-hard. Check for both before reaching for a table.

7.10.2 Measured

The distinction is not a matter of taste; it can be counted. **Listing 7.7** runs each of three recursions and records two numbers: how many calls were made, and how many *distinct* subproblems those calls represent.

Listing 7.7 Counting calls against distinct subproblems for Merge Sort, naive Fibonacci and plain matrix-chain recursion — an empirical test for overlapping subproblems. Compile with `g++ -std=c++17 -O2 -o overlap overlap.cpp`.

```
// overlap.cpp - the line between divide and conquer and dynamic programming,
// measured: how many calls are made, and how many DIFFERENT subproblems there are.
#include <iostream>
#include <iomanip>
#include <set>
#include <vector>
#include <utility>
using namespace std;

long long mergeCalls; set<pair<int,int>> mergeSub;      // subproblem = the range (lo, hi)
void mergeSortShape(int lo, int hi) {
    ++mergeCalls; mergeSub.insert({lo, hi});
    if (hi - lo <= 1) return;
    int mid = lo + (hi - lo) / 2;
    mergeSortShape(lo, mid);
    mergeSortShape(mid, hi);
}

long long fibCalls; set<int> fibSub;                     // subproblem = the argument n
void fibShape(int n) {
    ++fibCalls; fibSub.insert(n);
    if (n <= 1) return;
    fibShape(n - 1);
    fibShape(n - 2);
}

long long mcmCalls; set<pair<int,int>> mcmSub;           // subproblem = the chain (i, j)
void mcmShape(int i, int j) {
    ++mcmCalls; mcmSub.insert({i, j});
    if (i == j) return;
    for (int k = i; k < j; ++k) { mcmShape(i, k); mcmShape(k + 1, j); }
}

int main() {
    cout << left << setw(22) << "algorithm" << setw(8) << "size"
         << setw(16) << "calls made" << setw(16) << "distinct subs"
         << setw(12) << "ratio" << "\n";
    for (int n : {8, 16, 24}) {
        mergeCalls = 0; mergeSub.clear(); mergeSortShape(0, n);
        cout << left << setw(22) << "Merge Sort" << setw(8) << n
             << setw(16) << mergeCalls << setw(16) << mergeSub.size()
             << setw(12) << fixed << setprecision(2)
             << (double)mergeCalls / mergeSub.size() << "\n";
    }
    for (int n : {8, 16, 24}) {
        fibCalls = 0; fibSub.clear(); fibShape(n);
```

Listing 7.7 (Continued) *Counting calls against distinct subproblems for Merge Sort, naive Fibonacci and plain matrix-chain recursion — an empirical test for overlapping subproblems. Compile with g++ -std=c++17 -O2 -o overlap overlap.cpp.*

```
        cout << left << setw(22) << "naive Fibonacci" << setw(8) << n
             << setw(16) << fibCalls << setw(16) << fibSub.size()
             << setw(12) << fixed << setprecision(2)
             << (double)fibCalls / fibSub.size() << "\n";
    }
    for (int n : {4, 8, 12}) {
        mcmCalls = 0; mcmSub.clear(); mcmShape(1, n);
        cout << left << setw(22) << "matrix chain (plain)" << setw(8) << n
             << setw(16) << mcmCalls << setw(16) << mcmSub.size()
             << setw(12) << fixed << setprecision(2)
             << (double)mcmCalls / mcmSub.size() << "\n";
    }
    cout << "\nA ratio near 1 means no subproblem is ever revisited: divide and conquer.\n"
            "A ratio that grows without bound means the recursion is redoing work: use
DP.\n";
    return 0;
}
```

Its output:

```
algorithm             size    calls made      distinct subs   ratio
Merge Sort            8       15              15              1.00
Merge Sort            16      31              31              1.00
Merge Sort            24      47              47              1.00
naive Fibonacci       8       67              9               7.44
naive Fibonacci       16      3193            17              187.82
naive Fibonacci       24      150049          25              6001.96
matrix chain (plain)  4       27              10              2.70
matrix chain (plain)  8       2187            36              60.75
matrix chain (plain)  12      177147          78              2271.12

A ratio near 1 means no subproblem is ever revisited: divide and conquer.
A ratio that grows without bound means the recursion is redoing work: use DP.
```

Read the ratio column. For **Merge Sort** it is exactly 1.00 at every size: every call is a new subproblem, nothing is ever repeated. Memoizing Merge Sort would add a table that is written once per entry and read never.

For **naive Fibonacci** the ratio is 7.44, then 187.82, then 6002 — it grows without bound, and the distinct-subproblem column barely moves (9, 17, 25 — that is $n + 1$).

For **plain matrix chain** the pattern is the same: at $n = 12$ there are 177 147 calls covering only 78 distinct subchains, and $78 = 12 \cdot 13/2$, exactly the $n(n + 1)/2$ of §7.6.6.

7.10.3 The decision procedure

HOW TO DECIDE, IN PRACTICE. Given a recursive algorithm: **1.** Name the subproblem precisely — what tuple of values identifies it? **2.** Count the distinct values that tuple can take. If that count is exponential, dynamic programming will not save you; look for a greedy algorithm (Chapter 9), a different formulation, or accept exponential time. **3.** If the count is polynomial but the recursion makes far more calls than that, you have overlapping subproblems. Memoize or tabulate. **4.** If the count is polynomial and the recursion makes about that many calls, the subproblems are already independent. You have divide and conquer; leave it alone. Step 3 versus step 4 is exactly the ratio column of **Listing 7.7**, and you can measure it in five minutes with a `set` and a counter.

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	$T(n) = aT(n/b) + f(n)$	$\text{value}(s) = \text{opt over choices}$
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

7.11 Profitina in Practice

The AI Visibility Engine behind **app.profitina.com** scans a website and scores how ready it is to be cited by an AI answer engine. Several parts of that pipeline are dynamic programs of exactly the kinds in this chapter, and one of them is reason enough to master the chapter.

Near-duplicate page detection. A large brand site frequently publishes the same article at several URLs — a category path, a campaign path, an AMP path. Scoring all of them separately wastes crawl budget and distorts the site's score. Deciding whether two pages are "the same article, slightly edited" is a string-similarity question, and edit-distance and longest-common-subsequence dynamic programs are the classical tools for it. Here the $\Theta(\min(m, n))$-space variant of §7.9.10 is not an optimization but a requirement: two long articles produce a table with hundreds of millions of cells, and the service that answers the question has to run inside a bounded memory budget alongside everything else on the host.

The technical assistant's suggestions. When the Content Engine proposes a rewrite of a page's heading structure, the user interface must show *what changed* rather than dumping two blocks of text side by side. That difference view is the reconstruction step of §7.9.9 — the arrow table walked backward — and it is the reason the b table is worth its memory in that code path but not in the duplicate detector.

Chained transformations in the scoring pipeline. Scoring a page applies a sequence of transformations to a batch of extracted signals, several of which are linear maps applied to a matrix of observations. The order in which a chain of such maps is composed changes the size of the intermediate results and hence the arithmetic performed, which is precisely the matrix-chain problem of §7.6. The chains are short — the $\Theta(n^3)$ of **Matrix-Chain-Order** is nothing at n under ten — and the plan is computed once and cached, not recomputed per page.

Re-scan scheduling. Each brand is re-scanned at a cadence set by its tier. Fitting the day's scans into a bounded number of worker slots, so that no tier misses its promised cadence, is a scheduling problem with optimal substructure. As always: the app advises, and the user decides — a scan plan is a recommendation, never a guarantee about a score.

None of the above touches personal data. Where a signal does — a contact page carrying names, for instance — UU PDP 27/2022 applies, and the rule the codebase follows is simply that no personal field ever enters a similarity computation in the first place.

7.11.1 The worked case study: Profitina Laundry

Profitina Laundry is the teaching case study of this book — a small laundry in Surabaya with a counter, a price list, and a stack of handwritten slips at the end of each day.

The problem is dull and universal. The price list has canonical names — *Cuci Setrika Reguler*, *Bed Cover Besar*, *Sepatu Kanvas*. What the counter staff write on the slips is whatever fits: `cuci kring expres`, `setrka saja`, `bedcover besar`, `cuci kering reg`. Before the day can be totaled, each slip must be matched to a catalog entry.

Exact string comparison fails immediately — `bedcover besar` is not equal to `Bed Cover Besar`. What we need is a *measure of similarity*, and §7.9 has given us one. Define Equation (7.15):

$$\text{similarity}(u, v) = \frac{|\text{LCS}(u, v)|}{\max(|u|, |v|)}, \tag{7.15}$$

where $|u|$ means the length of string u, after both strings have been folded to lower case with spaces and punctuation removed. The numerator counts characters that appear in both, in order; the denominator normalizes so that the score lies between 0 (nothing in common) and 1 (identical). **Listing 7.8** is the whole system.

Listing 7.8 Profitina Laundry: matching handwritten slips to the price list with LCS similarity, in $\Theta(\min(m, n))$ space. Compile with `g++ -std=c++17 -O2 -o laundry laundry.cpp`.

```
// laundry.cpp - Profitina Laundry: reconciling what the counter staff wrote down
// with the canonical price list, using the LCS of this chapter as the measure of
// similarity. Nothing here touches personal data: only item names and prices.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
#include <chrono>
using namespace std;
using namespace std::chrono;

// Theta(mn) time, Theta(min(m,n)) space: we only need the LENGTH here.
int lcsLen(const string& A, const string& B) {
    const string& X = A.size() >= B.size() ? A : B;      // long string drives the outer
loop
    const string& Y = A.size() >= B.size() ? B : A;      // short string sizes the rows
    int m = (int)X.size(), n = (int)Y.size();
    vector<int> prev(n + 1, 0), cur(n + 1, 0);
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j)
            cur[j] = (X[i-1] == Y[j-1]) ? prev[j-1] + 1 : max(prev[j], cur[j-1]);
        swap(prev, cur);
    }
    return prev[n];
}

string normalize(const string& s) {                       // fold case, drop everything
else
    string t;
    for (char ch : s) if (isalnum((unsigned char)ch)) t += (char)tolower((unsigned
char)ch);
    return t;
}

struct Item { string name; long long pricePerKg; };

int main() {
    // The canonical price list. Prices are in rupiah per kilogram.
    vector<Item> catalog = {
        {"Cuci Kering Reguler",  7000},
        {"Cuci Kering Express",  12000},
        {"Cuci Setrika Reguler", 10000},
        {"Cuci Setrika Express", 16000},
        {"Setrika Saja",          6000},
        {"Bed Cover Besar",      35000},
        {"Selimut Tebal",        28000},
        {"Sepatu Kanvas",        25000}
    };
```

Listing 7.8 (Continued) *Profitina Laundry: matching handwritten slips to the price list with LCS similarity, in \Theta(\min(m,n)) space. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
    // What was actually written on the day's slips: abbreviated, misspelled, mixed case.
    vector<pair<string,double>> slips = {
        {"cuci setrika reguler", 3.5},
        {"cuci kring expres",    2.0},
        {"setrka saja",          1.5},
        {"bedcover besar",       1.0},
        {"cuci kering reg",      4.25},
        {"sepatu knvas",         1.0}
    };

    cout << "PROFITINA LAUNDRY - reconciling the day's slips against the price list\n\n";
    cout << left << setw(22) << "written on the slip" << setw(24) << "matched to"
         << setw(9) << "score" << setw(7) << "kg" << right << setw(12) << "rupiah" << "\n";
    cout << string(74, '-') << "\n";

    long long total = 0;
    auto t0 = steady_clock::now();
    for (auto& [written, kg] : slips) {
        string w = normalize(written);
        int bestIdx = -1; double bestScore = -1;
        for (int i = 0; i < (int)catalog.size(); ++i) {
            string cn = normalize(catalog[i].name);
            int L = lcsLen(w, cn);
            // Similarity in [0,1]: the LCS length against the longer of the two names.
            double score = (double)L / max(w.size(), cn.size());
            if (score > bestScore) { bestScore = score; bestIdx = i; }
        }
        long long amount = (long long)(catalog[bestIdx].pricePerKg * kg + 0.5);
        total += amount;
        cout << left << setw(22) << written << setw(24) << catalog[bestIdx].name
             << setw(9) << fixed << setprecision(3) << bestScore
             << setw(7) << setprecision(2) << kg << right << setw(12) << amount << "\n";
    }
    auto t1 = steady_clock::now();
    cout << string(74, '-') << "\n";
    cout << left << setw(62) << "TOTAL" << right << setw(12) << total << "\n";
    cout << "\nAll " << slips.size() << " slips matched against " << catalog.size()
         << " catalog entries in "
         << fixed << setprecision(3) << duration<double,milli>(t1-t0).count() << " ms.\n";
```

Listing 7.8 (Continued) *Profitina Laundry: matching handwritten slips to the price list with LCS similarity, in \Theta(\min(m,n)) space. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
    // A slip that should NOT be matched: the score stays low and a human must look.
    string junk = "titip motor sebentar";
    string jn = normalize(junk);
    double worst = 0; string which;
    for (auto& it : catalog) {
        string cn = normalize(it.name);
        double s = (double)lcsLen(jn, cn) / max(jn.size(), cn.size());
        if (s > worst) { worst = s; which = it.name; }
    }
    cout << "\nGuard rail: \"" << junk << "\" scores only "
         << fixed << setprecision(3) << worst << " against its closest entry ("
         << which << ").\nBelow a threshold of 0.600 the app refuses to guess and asks a
person.\n";
    return 0;
}
```

Its output:

```
PROFITINA LAUNDRY - reconciling the day's slips against the price list

written on the slip   matched to              score    kg           rupiah
---------------------------------------------------------------------------
cuci setrika reguler  Cuci Setrika Reguler    1.000    3.50          35000
cuci kring expres     Cuci Kering Express     0.882    2.00          24000
setrka saja           Setrika Saja            0.909    1.50           9000
bedcover besar        Bed Cover Besar         1.000    1.00          35000
cuci kering reg       Cuci Kering Reguler     0.765    4.25          29750
sepatu knvas          Sepatu Kanvas           0.917    1.00          25000
---------------------------------------------------------------------------
TOTAL                                                                157750

All 6 slips matched against 8 catalog entries in 0.058 ms.

Guard rail: "titip motor sebentar" scores only 0.389 against its closest entry (Cuci
Setrika Reguler).
Below a threshold of 0.600 the app refuses to guess and asks a person.
```

Every slip matched correctly. Note four things.

`bedcover besar` **scores a perfect** 1.000 against *Bed Cover Besar*, because normalization removed the spaces and the two strings became character-for-character identical. Cheap preprocessing before an expensive algorithm is almost always worth it.

`cuci kring expres` **scores** 0.882 despite three separate misspellings, because LCS does not care where the missing letters were — it counts what survived, in order.

`cuci kering reg` **scores only** 0.765, the lowest of the six, because the abbreviation `reg` threw away six of the nine characters of *Reguler*. It still matched correctly, but the margin is thinner, and this is the row a human should be shown if the shop ever disputes a bill.

The guard rail matters most of all. A slip reading *titip motor sebentar* — "park the motorbike a moment", not a laundry service at all — scores 0.389 against its nearest catalog entry. The system is built to refuse below 0.600 rather than to guess. That is the house rule, applied in code: *the app advises; the user decides*. An algorithm that always returns its best guess, however bad, will confidently bill a customer for a service nobody ordered.

On cost: six slips against eight catalog entries is forty-eight LCS computations on strings of about twenty characters, and the whole reconciliation took 0.058 ms. A day's four hundred slips would take about four milliseconds. The $\Theta(mn)$ of §7.9.10 is comfortably affordable here — but the same code run against whole documents, as in the near-duplicate detector above, is exactly where the two-row variant stops being an elegance and starts being the difference between working and not.

7.12 Common Mistakes, and How to Avoid Them

1. Believing recursion is the problem. It is not. Repetition is. The memoized version in **Listing 7.2** is still recursive and is 886 000 times faster than the naive one. Do not rewrite a recursion as a loop hoping for speed; find out first whether it is repeating itself.

2. Filling the table in the wrong order. In matrix chains, looping `for i` then `for j` reads cells that have not been written yet. Loop over the chain **length**. The test is mechanical: for the cell you are about to compute, list everything it depends on and check each one has a smaller value of the loop variable.

3. Getting k's range wrong. The matrix-chain minimum runs over $i \leq k < j$. Writing $k \leq j$ puts the entire chain on the left and nothing on the right, and the program computes $M[i,j]$ from $M[i,j]$ — which is either an infinite loop or a wrong answer, depending on your table's initial values.

4. Confusing subsequence with substring. A subsequence may have gaps; a substring may not. If your LCS of "ACGGTTA" and "CGTAT" comes out as 3, you have probably written the longest common *substring* algorithm, which is a different (and simpler) recurrence.

5. Using a sentinel a real answer could equal. If "not yet computed" is encoded as 0 and some subproblem genuinely has answer 0, that subproblem is recomputed every single time and your memoization silently does nothing. Use ∞, -1, or a separate `computed` flag.

6. Forgetting the choice table and then needing the solution. The M table gives the optimal value; only c gives the bracketing, and only b gives the subsequence. Deciding afterward that you need the solution means running the whole thing again.

7. Off-by-one between string indices and table indices. The table has a row 0 and column 0 for empty prefixes, so x_i is `X[i-1]` in C++. Write that line once, carefully, and never touch it again.

8. Assuming optimal substructure. Prove it by cut-and-paste, and check the paste is *legal*. Longest simple path looks exactly like shortest path and has no optimal substructure at all (§7.5.4).

9. Blowing the stack with a deep memoized recursion. Top-down recursion on a chain of a million elements will overflow before it computes anything: measured in §7.4.5, the crash came between $n = 250\,000$ and $n = 300\,000$ on an 8 MiB stack. Bottom-up has no such limit.

10. Reporting the reversed answer. Reconstruction walks from the end of the strings backward, so characters come out in reverse. The recursive reconstruction in **Listing 7.6** recurses *before* appending, which fixes this automatically; an iterative version must reverse the string at the end.

11. Paying for a table you do not need. If only the value is required, forget the rows you will never read again — $\Theta(n) \to \Theta(1)$ for Fibonacci, $\Theta(mn) \to \Theta(\min(m,n))$ for LCS. At $m = n = 8000$ that was 488 MiB against 0.061 MiB, and the small one ran twelve times faster.

12. Reaching for dynamic programming when the subproblems do not overlap. Measure the ratio of calls to distinct subproblems, as **Listing 7.7** does. If it is 1.00, a memo table is pure overhead.

7.13 Chapter Summary

Divide and conquer divides a problem into **disjoint** subproblems, solves each recursively, and combines the results (**Figure 7.1**). It is the right discipline when the pieces are genuinely different, as in Merge Sort (**Figure 7.2**).

When the pieces are *not* different, a plain recursion recomputes the same answers exponentially often. The smallest example is Fibonacci (**Figure 7.3**), whose recursion tree (**Figure 7.4**) has $2F_{n+1} - 1$ nodes carrying only $n + 1$ distinct labels, giving a $\Theta(\phi^n)$ running time with $\phi = (1 + \sqrt{5})/2$ (**Figure 7.5**). Measured, that is 866 million calls and two seconds for F_{42}.

Two repairs fix it, and both are dynamic programming: **memoization**, which keeps the recursion but consults a table of answers already found; and **bottom-up tabulation**, which abandons

recursion and builds the answers smallest-first (**Figure 7.6**). Both make Fibonacci $\Theta(n)$. A sliding window makes it $\Theta(1)$ in space, and a matrix power makes it $\Theta(\lg n)$ in time (**Table 7.2**).

Dynamic programming applies to **optimization problems** (**Figure 7.7**) that have **optimal substructure** — proved by a cut-and-paste argument — and **overlapping subproblems**. The recipe has four steps (**Figure 7.15**, **Figure 7.16**, **Figure 7.17**): characterize, recursively define, compute, construct.

Matrix-chain multiplication asks for the cheapest bracketing of $A_1 \cdots A_n$, where A_i is $d_{i-1} \times d_i$ and one product costs $n \cdot m \cdot k$ (**Figure 7.8**). The bracketing matters enormously (**Figure 7.9**): for the four matrices of §7.6.3 the spread was 1 400 against 41 200. There are Catalan-many bracketings, so brute force is out (**Listing 7.3**). Splitting at the outermost bracket (**Figure 7.10**) gives the recurrence Equation (7.10) (**Figure 7.11**)

$$M(i,j) = \min_{i \le k < j} \{M(i,k) + M(k+1,j) + d_{i-1}d_k d_j\},$$

with only $\Theta(n^2)$ distinct subproblems (**Figure 7.12**). **Matrix-Chain-Order** (**Pseudocode 7.5**) fills the table by increasing chain length in $\Theta(n^3)$ time — exactly $(n^3 - n)/6$ innermost iterations — and $\Theta(n^2)$ space (**Figure 7.14**), and **Print-Optimal-Parens** reads the bracketing out of the c table (**Figure 7.13**, **Table 7.3**, **Table 7.4**). Worked by hand on $d = \langle 10,20,3,5,30 \rangle$, the answer is 1 950 with grouping $((A_1A_2)(A_3A_4))$, confirmed three independent ways.

The same recurrence written as memoized recursion is **Lookup-Chain** (**Pseudocode 7.7**). Both forms are $\Theta(n^3)$; bottom-up wins on constants, memoization wins on convenience (**Table 7.5**).

The **longest common subsequence** (**Figure 7.18**) has optimal substructure in three cases (**Figure 7.19**), giving the recurrence Equation (7.14) (**Figure 7.20**)

$$c[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ c[i-1,j-1] + 1 & x_i = y_j \\ \max\{c[i,j-1], c[i-1,j]\} & x_i \neq y_j \end{cases}$$

LCS-Length (**Pseudocode 7.9**) fills it in $\Theta(mn)$, recording arrows in b. On the strings of **Figure 7.21** the tables are **Table 7.6** and **Table 7.7**, the answer is 4, and following the arrows back gives "CGTT". Dropping b and keeping two rows reduces the space to $\Theta(\min(m,n))$.

The line between the two disciplines is **overlapping versus independent** subproblems, and it can be measured directly (**Listing 7.7**, **Table 7.8**): Merge Sort's ratio of calls to distinct subproblems is exactly 1.00; Fibonacci's is 6002 at $n = 24$.

7.13.1 Looking ahead

Chapter 8, *Evaluation and Discussion II*, works problems drawn from Chapters 5 to 7 in full, and its revision drills return to the two hallmarks of dynamic programming — overlapping subproblems and optimal substructure — that this chapter has established.

Chapter 9 introduces **greedy algorithms**, the other great technique for optimization problems. A greedy algorithm also needs optimal substructure, but it needs something stronger too: the *greedy-choice property*, which says that a locally best choice made now is part of some globally optimal solution. When that property holds you may take the obvious choice and never look back — no table, no minimization over k, often $\Theta(n\lg n)$ instead of $\Theta(n^3)$. When it fails — and the matrix-chain problem is a good example of a problem where it fails, since no simple local rule picks the right split — you are back to dynamic programming. Knowing which of the two you are looking at is the skill Chapter 9 builds, and it rests entirely on the notion of optimal substructure defined in §7.5.2 of this chapter.

7.14 Exercises

7.1 Write out F_0 through F_{12}, showing each addition. Then compute, using $C(n) = 2F_{n+1} - 1$, how many calls the naive recursion of **Listing 7.1** makes for $n = 10$ and for $n = 20$.

7.2 Prove that the naive Fibonacci recursion tree for `fib(n)` has exactly F_{n+1} leaves, of which F_n are `F(1)` and F_{n-1} are `F(0)` (for $n \geq 1$). Deduce that the value returned, F_n, is literally the number of `F(1)` leaves.

7.3 Trace the sliding-window iteration of §7.4.3 for $n = 8$, one line per iteration, in the format of **Table 7.1**.

7.4 In the output of **Listing 7.1**, the ratio of successive call counts (two steps apart) is 2.618 at every size. Explain exactly why, and predict the ratio if the table had stepped by 1 and by 3 instead of 2.

7.5 Let A be 5×10, B be 10×2, C be 2×8. Compute the cost of $(AB)C$ and of $A(BC)$, showing each multiplication. Which wins, and by what factor?

7.6 Using $P(1) = 1$ and $P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$, compute $P(6)$, $P(7)$ and $P(8)$ by hand.

7.7 Fill the M and c tables by hand for the chain $d = \langle 30,35,15,5,10,20,25 \rangle$ ($n = 6$). Give the optimal cost and, using **Print-Optimal-Parens**, the optimal bracketing.

7.8 Using the exact formula $(n^3 - n)/6$ derived in §7.6.10, state how many times the innermost statement of **Matrix-Chain-Order** executes for $n = 6$ and for $n = 10$. Verify the $n = 6$ figure against your work in Exercise 7.7.

7.9 Consider this greedy heuristic: *repeatedly perform the cheapest available adjacent multiplication.* Run it on $d = \langle 10,20,3,5,30 \rangle$ and compare with the optimum of 1 950 found in §7.6.8. What does this show?

7.10 You are handed only this c table for a chain of five matrices:

Table 7.9 The c table for Exercise 7.10. Reconstruct the bracketing it encodes.

	j=2	j=3	j=4	j=5
$i = 1$	1	1	3	3
$i = 2$	—	2	3	3
$i = 3$	—	—	3	4
$i = 4$	—	—	—	4

Run **Print-Optimal-Parens** by hand on **Table 7.9** and write down the bracketing.

7.11 Compute the full c table for $X = $ "ABCBDAB" and $Y = $ "BDCABA" by hand. Give the LCS length, and reconstruct one LCS using the $\geq$ tie-breaking rule of **Pseudocode 7.9**.

7.12 For $X = $ "AGCAT" and $Y = $ "GAC", build the c table and find the LCS length. Then find *all* the longest common subsequences, not just one.

7.13 Prove two properties of any LCS table: (a) $c[i,j] \geq c[i-1,j]$ and $c[i,j] \geq c[i,j-1]$; (b) $c[i,j] \leq c[i-1,j]+1$ and $c[i,j] \leq c[i,j-1]+1$. What do these say about how the numbers can change as you walk across a row?

7.14 The two-row variant of §7.9.10 computes the LCS *length* in $\Theta(\min(m,n))$ space but cannot reconstruct the subsequence. Explain precisely why not. Then read about Hirschberg's algorithm and state, in two sentences, how it recovers the subsequence in linear space, and what it costs.

7.15 The **edit distance** between two strings is the fewest single-character insertions, deletions and substitutions that turn one into the other. Apply the four steps of §7.8 to derive its recurrence. Then compute the edit distance between "kitten" and "sitting" by hand.

7.16 Show that the **longest strictly increasing subsequence** of a sequence A of n distinct numbers is the LCS of A with a sorted copy of A. What running time does that give, and is it the best known?

7.17 **Listing 7.7** reports a calls-to-distinct-subproblems ratio of exactly 1.00 for Merge Sort at every size. Prove that the ratio is always exactly 1, and state what a memo table would cost if you added one anyway.

7.18 *(Design task.)* Profitina Laundry has bags of capacity 3 kg, 5 kg and 8 kg. Every bag used costs the same flat fee, and a bag must be filled *exactly* to its capacity. Apply the four steps of §7.8 to find the fewest bags that hold exactly W kilograms. Solve it by hand for every W from 0 to 13, write the program, and show that a greedy "use the biggest bag that fits" rule fails.

7.15 Solutions to the Exercises

7.1 $F_0 = 0$; $F_1 = 1$; $F_2 = 1+0 = 1$; $F_3 = 1+1 = 2$; $F_4 = 2+1 = 3$; $F_5 = 3+2 = 5$; $F_6 = 5+3 = 8$; $F_7 = 8+5 = 13$; $F_8 = 13+8 = 21$; $F_9 = 21+13 = 34$; $F_{10} = 34+21 = 55$; $F_{11} = 55+34 = 89$; $F_{12} = 89+55 = 144$.

For $n = 10$: $C(10) = 2F_{11} - 1 = 2(89) - 1 = 177$ calls to compute the number 55. For $n = 20$: $F_{21} = 10\,946$, so $C(20) = 2(10\,946) - 1 = 21\,891$. That is exactly the figure **Listing 7.1** printed for $n = 20$.

7.2 Let $L(n)$ be the number of leaves, $L_1(n)$ the number of `F(1)` leaves and $L_0(n)$ the number of `F(0)` leaves.

Leaves. A call with $n \leq 1$ is itself a leaf, so $L(0) = L(1) = 1$. A call with $n \geq 2$ is internal, and its leaves are exactly the leaves of its two children, so $L(n) = L(n-1) + L(n-2)$. That is the Fibonacci recurrence with $L(0) = L(1) = 1$. Since $F_1 = F_2 = 1$, the sequence L is F shifted by one: $L(n) = F_{n+1}$. (Formally, by strong induction: it holds at $n = 0,1$, and $L(n) = L(n-1) + L(n-2) = F_n + F_{n-1} = F_{n+1}$.)

The split. L_1 satisfies $L_1(0) = 0$, $L_1(1) = 1$, $L_1(n) = L_1(n-1) + L_1(n-2)$ — the Fibonacci recurrence with Fibonacci's own initial values — so $L_1(n) = F_n$. Similarly $L_0(0) = 1$, $L_0(1) = 0$, giving $L_0(n) = F_{n-1}$. Check: $F_n + F_{n-1} = F_{n+1} = L(n)$ ✓.

The deduction. Every internal node returns the sum of its children, and every leaf returns its own value — 1 for an `F(1)` leaf, 0 for an `F(0)` leaf. So the root's value is the sum of all leaf values, which is $1 \cdot L_1(n) + 0 \cdot L_0(n) = F_n$. The algorithm computes F_n by literally adding F_n ones together, one at a time. For $n = 6$: eight `F(1)` leaves and five `F(0)` leaves, and $F_6 = 8$ — exactly what **Listing 7.1** printed and what **Figure 7.4** shows.

7.3 The trace is **Table 7.10**.

Table 7.10 The sliding window computing $F_8 = 21$. Two variables, seven iterations, no array.

i	prev before	cur before	next	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
7	5	8	13	8	13
8	8	13	21	13	21

7.4 The call count is $C(n) = 2F_{n+1} - 1$, and $F_n \approx \phi^n/\sqrt{5}$, so $C(n) \approx (2/\sqrt{5})\phi^{n+1}$. Then

$$\frac{C(n+2)}{C(n)} \approx \frac{\phi^{n+3}}{\phi^{n+1}} = \phi^2 = 2.6180339\ldots$$

which matches the measured 2.618 to every digit printed. Stepping by 1 would give $\phi^1 = 1.618$; stepping by 3 would give $\phi^3 = 4.236$. (Note $\phi^2 = \phi + 1$, since ϕ solves $x^2 = x + 1$ — so the ratio is just $1.618 + 1$.)

7.5

$(AB)C$: AB costs $5 \times 10 \times 2 = 100$, giving a 5×2 matrix; then $(5 \times 2)(2 \times 8)$ costs $5 \times 2 \times 8 = 80$. Total $100 + 80 = \mathbf{180}$.

$A(BC)$: BC costs $10 \times 2 \times 8 = 160$, giving a 10×8 matrix; then $(5 \times 10)(10 \times 8)$ costs $5 \times 10 \times 8 = 400$. Total $160 + 400 = \mathbf{560}$.

$(AB)C$ wins by a factor of $560/180 = 3.1$. The reason is the same as in §7.6.3: $(AB)C$ collapses the inner dimension from 10 to 2 immediately and never builds a 10×8 intermediate.

7.6 With $P(1) = P(2) = 1$, $P(3) = 2$, $P(4) = 5$, $P(5) = 14$ from §7.6.4:

$P(6) = P_1P_5 + P_2P_4 + P_3P_3 + P_4P_2 + P_5P_1 = 14 + 5 + 4 + 5 + 14 = \mathbf{42}$.

$P(7) = P_1P_6 + P_2P_5 + P_3P_4 + P_4P_3 + P_5P_2 + P_6P_1 = 42 + 14 + 10 + 10 + 14 + 42 = \mathbf{132}$.

$P(8) = P_1P_7 + P_2P_6 + P_3P_5 + P_4P_4 + P_5P_3 + P_6P_2 + P_7P_1 = 132 + 42 + 28 + 25 + 28 + 42 + 132 = \mathbf{429}$.

These agree with the Catalan numbers printed by **Listing 7.5**.

7.7 With $d = \langle 30,35,15,5,10,20,25 \rangle$, so A_1 is 30×35, A_2 is 35×15, A_3 is 15×5, A_4 is 5×10, A_5 is 10×20, A_6 is 20×25.

Length 2. $M[1,2] = 30 \cdot 35 \cdot 15 = 15\,750$ ($c = 1$); $M[2,3] = 35 \cdot 15 \cdot 5 = 2\,625$ ($c = 2$); $M[3,4] = 15 \cdot 5 \cdot 10 = 750$ ($c = 3$); $M[4,5] = 5 \cdot 10 \cdot 20 = 1\,000$ ($c = 4$); $M[5,6] = 10 \cdot 20 \cdot 25 = 5\,000$ ($c = 5$).

Length 3. $M[1,3]$: $k = 1$ gives $0 + 2625 + 30 \cdot 35 \cdot 5 = 2625 + 5250 = 7\,875$; $k = 2$ gives $15750 + 0 + 30 \cdot 15 \cdot 5 = 15750 + 2250 = 18\,000$. Min 7 875, $c = 1$. $M[2,4]$: $k = 2$: $0 + 750 + 35 \cdot 15 \cdot 10 = 750 + 5250 = 6\,000$; $k = 3$: $2625 + 0 + 35 \cdot 5 \cdot 10 = 2625 + 1750 = 4\,375$. Min 4 375, $c = 3$. $M[3,5]$: $k = 3$: $0 + 1000 + 15 \cdot 5 \cdot 20 = 1000 + 1500 = 2\,500$; $k = 4$: $750 + 0 + 15 \cdot 10 \cdot 20 = 750 + 3000 = 3\,750$. Min 2 500, $c = 3$. $M[4,6]$: $k = 4$: $0 + 5000 + 5 \cdot 10 \cdot 25 = 5000 + 1250 = 6\,250$; $k = 5$: $1000 + 0 + 5 \cdot 20 \cdot 25 = 1000 + 2500 = 3\,500$. Min 3 500, $c = 5$.

Length 4. $M[1,4]$: $k = 1$: $0 + 4375 + 30 \cdot 35 \cdot 10 = 4375 + 10500 = 14\,875$; $k = 2$: $15750 + 750 + 30 \cdot 15 \cdot 10 = 16500 + 4500 = 21\,000$; $k = 3$: $7875 + 0 + 30 \cdot 5 \cdot 10 = 7875 + 1500 = 9\,375$. Min 9 375, $c = 3$. $M[2,5]$: $k = 2$: $0 + 2500 + 35 \cdot 15 \cdot 20 = 2500 + 10500 = 13\,000$; $k = 3$: $2625 + 1000 + 35 \cdot 5 \cdot 20 = 3625 + 3500 = 7\,125$; $k = 4$: $4375 + 0 + 35 \cdot 10 \cdot 20 = 4375 + 7000 = 11\,375$. Min 7 125, $c = 3$. $M[3,6]$: $k = 3$: $0 + 3500 + 15 \cdot 5 \cdot 25 = 3500 + 1875 = 5\,375$; $k = 4$: $750 + 5000 + 15 \cdot 10 \cdot 25 = 5750 + 3750 = 9\,500$; $k = 5$: $2500 + 0 + 15 \cdot 20 \cdot 25 = 2500 + 7500 = 10\,000$. Min 5 375, $c = 3$.

Length 5. $M[1,5]$: $k = 1$: $0 + 7125 + 30 \cdot 35 \cdot 20 = 7125 + 21000 = 28\,125$; $k = 2$: $15750 + 2500 + 30 \cdot 15 \cdot 20 = 18250 + 9000 = 27\,250$; $k = 3$: $7875 + 1000 + 30 \cdot 5 \cdot 20 = 8875 + 3000 = 11\,875$; $k = 4$: $9375 + 0 + 30 \cdot 10 \cdot 20 = 9375 + 6000 = 15\,375$. Min 11 875, $c = 3$. $M[2,6]$: $k = 2$: $0 + 5375 + 35 \cdot 15 \cdot 25 = 5375 + 13125 = 18\,500$; $k = 3$: $2625 + 3500 + 35 \cdot 5 \cdot 25 = 6125 + 4375 = 10\,500$; $k = 4$: $4375 + 5000 + 35 \cdot 10 \cdot 25 = 9375 + 8750 = 18\,125$; $k = 5$: $7125 + 0 + 35 \cdot 20 \cdot 25 = 7125 + 17500 = 24\,625$. Min 10 500, $c = 3$.

The finished M table is **Table 7.11**.

Length 6. $M[1,6]$: $k = 1$: $0 + 10500 + 30 \cdot 35 \cdot 25 = 10500 + 26250 = 36\,750$; $k = 2$: $15750 + 5375 + 30 \cdot 15 \cdot 25 = 21125 + 11250 = 32\,375$; $k = 3$: $7875 + 3500 + 30 \cdot 5 \cdot 25 = 11375 + 3750 = 15\,125$; $k = 4$: $9375 + 5000 + 30 \cdot 10 \cdot 25 = 14375 + 7500 = 21\,875$; $k = 5$: $11875 + 0 + 30 \cdot 20 \cdot 25 = 11875 + 15000 = 26\,875$. Min **15 125**, $c = 3$.

Table 7.11 The M table for Exercise 7.7. The c table is $c[1,2] = 1$, $c[1,3] = 1$, $c[1,4] = 3$, $c[1,5] = 3$, $c[1,6] = 3$; $c[2,3] = 2$, $c[2,4] = 3$, $c[2,5] = 3$, $c[2,6] = 3$; $c[3,4] = 3$, $c[3,5] = 3$, $c[3,6] = 3$; $c[4,5] = 4$, $c[4,6] = 5$; $c[5,6] = 5$.

	j=1	j=2	j=3	j=4	j=5	j=6
$i = 1$	0	15750	7875	9375	11875	**15125**
$i = 2$	—	0	2625	4375	7125	10500
$i = 3$	—	—	0	750	2500	5375
$i = 4$	—	—	—	0	1000	3500
$i = 5$	—	—	—	—	0	5000
$i = 6$	—	—	—	—	—	0

Reconstruction: $c[1,6] = 3$ splits into (1,3) and (4,6). $c[1,3] = 1$ splits into A_1 and (2,3); $c[2,3] = 2$ gives (A_2A_3). So the left is $(A_1(A_2A_3))$. $c[4,6] = 5$ splits into (4,5) and A_6; $c[4,5] = 4$ gives (A_4A_5). So the right is $((A_4A_5)A_6)$. The optimal bracketing is

$$((A_1(A_2A_3))\,((A_4A_5)A_6)) \quad \text{at 15 125 multiplications.}$$

7.8 $n = 6$: $(6^3 - 6)/6 = (216 - 6)/6 = 35$. $n = 10$: $(1000 - 10)/6 = 165$. Checking the $n = 6$ figure against Exercise 7.7: lengths 2, 3, 4, 5, 6 contributed $5 \times 1 + 4 \times 2 + 3 \times 3 + 2 \times 4 + 1 \times 5 = 5 + 8 + 9 + 8 + 5 = 35$ ✓.

7.9 The available adjacent products cost $A_1A_2 = 10 \cdot 20 \cdot 3 = 600$, $A_2A_3 = 20 \cdot 3 \cdot 5 = 300$, $A_3A_4 = 3 \cdot 5 \cdot 30 = 450$.

Step 1. Cheapest is A_2A_3 at 300. Perform it; the chain becomes 10×20, 20×5, 5×30, i.e. $d = \langle 10,20,5,30 \rangle$. Running total 300. *Step 2.* Now the options are $A_1(A_2A_3) = 10 \cdot 20 \cdot 5 = 1\,000$ and $(A_2A_3)A_4 = 20 \cdot 5 \cdot 30 = 3\,000$. Cheapest is 1 000. Running total 1 300. *Step 3.* One product left: $10 \cdot 5 \cdot 30 = 1\,500$. **Total** 2 800**.**

The optimum is 1 950, so the greedy heuristic is 43% worse. Its answer, 2 800, is the bracketing $((A_1(A_2A_3))A_4)$, which **Listing 7.3** shows to be only the *third* cheapest of the five, behind 1 950 and 2 250. This shows that matrix-chain multiplication lacks the greedy-choice property: the locally cheapest first multiplication is not part of any optimal solution here. Compare Chapter 9, where problems that *do* have that property are treated.

7.10 `parens(1,5)`: $c[1,5] = 3$, so `"(" + parens(1,3) + parens(4,5) + ")"`. `parens(1,3)`: $c[1,3] = 1$, so `"(" + parens(1,1) + parens(2,3) + ")"` = `"(A1" + parens(2,3) + ")"`. `parens(2,3)`: $c[2,3] = 2$, giving `"(A2A3)"`. So the left part is `"(A1(A2A3))"`. `parens(4,5)`: $c[4,5] = 4$, giving `"(A4A5)"`. Answer: $((A_1(A_2A_3))(A_4A_5))$.

7.11 $X = $ "ABCBDAB" ($m = 7$), $Y = $ "BDCABA" ($n = 6$). The completed table is **Table 7.12**.

Table 7.12 The c table for Exercise 7.11. The LCS length is $c[7,6] = 4$.

	j=0	B	D	C	A	B	A
$i = 0$	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	**4**

Reconstructing from (7,6) with the $\geq$ rule: $x_7 =$ B, $y_6 =$ A — no match; above is $c[6,6] = 4$, left is $c[7,5] = 4$, and $4 \geq 4$ so go up to (6,6). There $x_6 =$ A $= y_6 =$ A — match, emit **A**, go to (5,5). There D vs B — no match; above $c[4,5] = 3$, left $c[5,4] = 2$, $3 \geq 2$ so up to (4,5). There B = B — match, emit **B**, go to (3,4). There C vs A — no match; above $c[2,4] = 1$, left $c[3,3] = 2$, $1 \geq 2$ false, so left to (3,3). There C = C — match, emit **C**, go to (2,2). There B vs D — no match; above $c[1,2] = 0$, left $c[2,1] = 1$, so left to (2,1). There B = B — match, emit **B**, go to (1,0). Stop.

Emitted A, B, C, B; reversed, the LCS is **"BCBA"**, length 4.

7.12 $X = $ "AGCAT", $Y = $ "GAC". The table is **Table 7.13**.

Table 7.13 The c table for Exercise 7.12. The LCS length is 2, and three distinct LCSs achieve it.

	j=0	G	A	C
$i = 0$	0	0	0	0
A	0	0	1	1
G	0	1	1	1
C	0	1	1	2
A	0	1	2	2
T	0	1	2	**2**

The LCS length is 2. There are three distinct longest common subsequences:

- **"AC"** — positions 1, 3 of X; positions 2, 3 of Y.
- **"GA"** — positions 2, 4 of X; positions 1, 2 of Y.
- **"GC"** — positions 2, 3 of X; positions 1, 3 of Y.

The algorithm returns only one of them, determined by the tie-breaking rule; with the $\geq$ rule of **Pseudocode 7.9** it returns "AC". This is why the definition in §7.9.2 says "*a* longest common subsequence".

7.13 (a) Any common subsequence of X_{i-1} and Y_j is also a common subsequence of X_i and Y_j (a subsequence of a prefix is a subsequence of the longer string). So the best for (i, j) is at least the best for $(i-1, j)$: $c[i, j] \geq c[i-1, j]$. The same argument with X and Y exchanged gives $c[i, j] \geq c[i, j-1]$.

(b) Let Z be an LCS of X_i and Y_j, of length $c[i, j]$. Delete its last character if that character is matched to x_i; the remainder is a common subsequence of X_{i-1} and Y_j, of length at least $c[i, j] - 1$. If the last character is *not* matched to x_i, then Z itself is already a common subsequence of X_{i-1} and Y_j. Either way $c[i-1, j] \geq c[i, j] - 1$, that is, $c[i, j] \leq c[i-1, j] + 1$. Symmetrically for the column.

Together they say: walking one step right or one step down, the value either stays the same or increases by exactly 1 — it never decreases and never jumps by 2. That is the invariant to check when debugging an LCS table by eye; in **Table 7.6** the row for $i = 5$ reads 0,1,2,3,3,3, and every step is $+1$ or $+0$ ✓.

7.14 *Why reconstruction fails.* Reconstruction walks *backward* through the choices, which means it needs the arrow (or the value) at cells in rows it has already passed. The two-row version overwrites each row as soon as the next one is computed, so by the time the last row is finished, rows 0 through $m - 2$ no longer exist. The information was not merely compressed; it was destroyed.

Hirschberg's algorithm. It splits X in half at row $m/2$, computes in linear space the LCS lengths of $X_{1..m/2}$ against every prefix of Y (a forward two-row pass) and of $X_{m/2+1..m}$ against every suffix of Y (a backward two-row pass), and picks the column j^* maximizing the sum — which is provably a point the optimal alignment passes through. It then recurses on the two halves $(X_{1..m/2}, Y_{1..j^*})$ and $(X_{m/2+1..m}, Y_{j^*+1..n})$. The cost is $\Theta(mn)$ time — about twice the plain version, because the work halves at each level and $mn + mn/2 + mn/4 + \cdots = 2mn$ — in $\Theta(\min(m, n))$ space. It is a divide-and-conquer algorithm wrapped around a dynamic program: both techniques of this chapter in one. See [3, §14.4] and [6].

7.15 *Step 1 — characterize.* Consider the last characters x_m and y_n. The choice is what the last edit operation is. If $x_m = y_n$ we may match them for free. Otherwise there are three possibilities: **delete** x_m (reducing to X_{m-1} vs Y_n), **insert** y_n (reducing to X_m vs Y_{n-1}), or **substitute** x_m by y_n (reducing to X_{m-1} vs Y_{n-1}). By the cut-and-paste argument, whichever operation the optimum uses, the remaining transformation must itself be optimal — otherwise replace it with a cheaper one and the total falls.

Step 2 — recurrence. Let $e[i,j]$ be the edit distance between X_i and Y_j. Then Equation (7.16) holds:

$$e[i,j] = \begin{cases} j & i = 0 \quad \text{(insert al} \\ i & j = 0 \quad \text{(delete al} \\ e[i-1,j-1] & x_i = y_j \\ 1 + \min\{e[i-1,j],\ e[i,j-1],\ e[i-1,j-1]\} & x_i \neq y_j \end{cases} \tag{7.16}$$

Subproblems: $(m+1)(n+1) = \Theta(mn)$ — polynomial ✓.

Steps 3 and 4. Fill row by row in $\Theta(mn)$ time and $\Theta(mn)$ space (or two rows if only the distance is wanted); record the winning operation to reconstruct the edit script.

"kitten" → "sitting". $m = 6, n = 7$; the filled table is **Table 7.14**.

Table 7.14 The edit-distance table for "kitten" and "sitting". The answer, $e[6,7] = 3$, is the bold cell.

	—	s	i	t	t	i	n	g
—	0	1	2	3	4	5	6	7
k	1	1	2	3	4	5	6	7
i	2	2	1	2	3	4	5	6
t	3	3	2	1	2	3	4	5
t	4	4	3	2	1	2	3	4
e	5	5	4	3	2	2	3	4
n	6	6	5	4	3	3	2	**3**

The distance is **3**: substitute k→s, substitute e→i, insert g.

7.16 Let S be the sorted copy of A. A common subsequence of A and S is a sequence of values that appears in A in its original order *and* in S in sorted order — that is, a subsequence of A that is increasing. Conversely any increasing subsequence of A is a subsequence of both. So the two sets coincide, and the longest members coincide.

The elements must be **distinct** for this to work. With repeats, a common subsequence of A and S may be non-decreasing rather than strictly increasing; the standard repair is to sort with duplicates removed.

Running time: $\Theta(n\lg n)$ for the sort plus $\Theta(n^2)$ for the LCS, so $\Theta(n^2)$. It is **not** the best known: a direct dynamic program with binary search over a table of smallest tail values solves the longest increasing subsequence in $\Theta(n\lg n)$; Halim *et al.* [7, §3.5] give that version with contest-ready code. The LCS reduction is a good way to *see* the problem, not the way to solve it.

7.17 A Merge Sort call is identified by the range (lo, hi) it sorts. The ranges produced form a binary tree in which each node's range is split into two halves that are then never re-produced by any other node: the children of (lo, hi) are (lo, mid) and (mid, hi), and these are subranges of (lo, hi) only. Two different nodes at the same depth have disjoint ranges; two nodes at different

depths have ranges that are either nested strictly or disjoint, and a strictly nested pair cannot be equal. Hence no range is ever produced twice, so distinct subproblems = calls and the ratio is exactly 1. (There are $2n - 1$ of each for n elements: the counts 15,31,47 in **Listing 7.7** are $2 \cdot 8 - 1$, $2 \cdot 16 - 1$ and $2 \cdot 24 - 1$.)

A memo table keyed on (lo, hi) would need one entry per call — $\Theta(n)$ entries, each holding a sorted array, so $\Theta(n \lg n)$ words in total — and every single lookup would miss. Pure cost, zero benefit.

7.18 *Step 1 — characterize.* The choice is *which bag to fill last*. If the last bag has capacity s, the rest of the load weighs $W - s$ and must be packed in the fewest bags. Cut and paste: if that sub-packing were not minimal, replace it with a smaller one and the total falls — contradiction. Optimal substructure ✓.

Step 2 — recurrence. Let $B[w]$ be the fewest bags holding exactly w kilograms, or ∞ if w cannot be made exactly. Then Equation (7.17) holds:

$$B[0] = 0, \qquad B[w] = 1 + \min_{s \in \{3,5,8\},\, s \le w} B[w - s]. \tag{7.17}$$

Subproblems: one per w from 0 to W, so $W + 1$ — polynomial in W ✓.

Step 3 — compute, bottom-up in increasing w, one line each:

- $B[0] = 0$.
- $B[1]$: no bag fits. ∞.
- $B[2]$: no bag fits. ∞.
- $B[3] = 1 + B[0] = 1$. Bags: 3.
- $B[4] = 1 + B[1] = \infty$.
- $B[5] = 1 + \min(B[2], B[0]) = 1 + 0 = 1$. Bags: 5.
- $B[6] = 1 + \min(B[3], B[1]) = 1 + 1 = 2$. Bags: 3+3.
- $B[7] = 1 + \min(B[4], B[2]) = \infty$.
- $B[8] = 1 + \min(B[5], B[3], B[0]) = 1 + 0 = 1$. Bags: 8.
- $B[9] = 1 + \min(B[6], B[4], B[1]) = 1 + 2 = 3$. Bags: 3+3+3.
- $B[10] = 1 + \min(B[7], B[5], B[2]) = 1 + 1 = 2$. Bags: 5+5.
- $B[11] = 1 + \min(B[8], B[6], B[3]) = 1 + 1 = 2$. Bags: 3+8.
- $B[12] = 1 + \min(B[9], B[7], B[4]) = 1 + 3 = 4$. Bags: 3+3+3+3.
- $B[13] = 1 + \min(B[10], B[8], B[5]) = 1 + 1 = 2$. Bags: 5+8.

Step 4 — construct: record, with each $B[w]$, the size s that attained the minimum, then peel off one bag at a time.

Listing 7.9 is the program.

Listing 7.9 The Profitina Laundry bag-packing dynamic program of Exercise 7.18, with a greedy comparison. Compile with `g++ -std=c++17 -O2 -o bags bags.cpp`.

```
// bags.cpp - Profitina Laundry: the fewest bags that hold exactly W kilograms,
// given bag capacities that must each be filled exactly. A four-step DP.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <climits>
using namespace std;

int main() {
    vector<int> size = {3, 5, 8};                  // available bag capacities, in kg
    int W = 13;
    const int INF = INT_MAX / 2;
    vector<int> B(W + 1, INF), choice(W + 1, -1); // B[w] = fewest bags holding exactly w
    B[0] = 0;                                      // zero kilograms needs zero bags
    for (int w = 1; w <= W; ++w)
        for (int s : size)
            if (s <= w && B[w - s] + 1 < B[w]) { B[w] = B[w - s] + 1; choice[w] = s; }

    cout << left << setw(6) << "w" << setw(10) << "bags" << "one optimal packing\n";
    for (int w = 0; w <= W; ++w) {
        cout << left << setw(6) << w;
        if (B[w] >= INF) { cout << setw(10) << "impossible" << "\n"; continue; }
        cout << setw(10) << B[w];
        for (int t = w; t > 0; t -= choice[t]) cout << choice[t] << (t - choice[t] ? " + "
: "");
        cout << "\n";
    }
    // Greedy, largest-first, for comparison.
    for (int w : {12, 13}) {
        int rem = w, used = 0; bool ok = true;
        for (int idx = (int)size.size() - 1; idx >= 0; --idx)
            while (rem >= size[idx]) { rem -= size[idx]; ++used; }
        if (rem != 0) ok = false;
        cout << "greedy largest-first on w=" << w << ": "
             << (ok ? to_string(used) + " bags" : "FAILS, " + to_string(rem) + " kg left
over")
             << "   (optimal = " << B[w] << ")\n";
    }
    return 0;
}
```

Its output:

```
w     bags      one optimal packing
0     0
1     impossible
2     impossible
3     1         3
4     impossible
5     1         5
6     2         3 + 3
7     impossible
8     1         8
9     3         3 + 3 + 3
10    2         5 + 5
11    2         3 + 8
12    4         3 + 3 + 3 + 3
13    2         5 + 8
greedy largest-first on w=12: FAILS, 1 kg left over   (optimal = 4)
greedy largest-first on w=13: 2 bags   (optimal = 2)
```

Every line matches the hand computation above.

Why greedy fails. On $W = 12$ the greedy rule takes an 8 kg bag, leaving 4 kg; it then takes a 3 kg bag, leaving 1 kg, which no bag can hold — greedy does not merely produce a worse answer, it

produces *no* answer, while the optimum is four 3 kg bags. Note also the delightful non-monotonicity: 12 kg needs four bags but 13 kg needs only two. Any rule that assumes "more weight needs more bags" is wrong, and only the table knows it.

References

[1] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[2] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton Univ. Press, 1957.

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

[4] R. P. Stanley, *Catalan Numbers*. Cambridge, U.K.: Cambridge Univ. Press, 2015.

[5] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[6] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, vol. 18, no. 6, pp. 341–343, Jun. 1975.

[7] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, 4th ed. Raleigh, NC, USA: Lulu Press, 2020.

Chapter 8

Evaluation and Discussion II: Sorting, Search Trees, and Dynamic Programming

This chapter gathers Chapters 5, 6 and 7 into one long workshop of solved problems. By the end of it you will have built a max-heap by hand, one swap at a time, and sorted the result with heapsort; traced Lomuto's partition one index at a time and counted, exactly, the comparisons Quicksort makes in its best and worst cases; written, from first principles, eight routines over binary search trees and AVL trees — a validator, two traversals, a balance checker, insertion, two rebalancers and deletion — each delivered as a complete C++ program that has been compiled and run; worked a second family of problems on red-black trees; filled a matrix-chain table and a longest-common-subsequence table cell by cell and read the answers back out of them; and solved a set of further problems on an order-statistic tree that forces every idea of the chapter to work at once. Where a problem has a tempting wrong answer, you will see it written down, run, and taken apart, because a wrong answer you understand teaches more than a right answer you memorized. Twenty-five quick checks, fourteen exercises and fully worked solutions close the chapter.

8.1 Where We Stand

Three chapters lie behind us. Chapter 5 taught two ways to sort that beat the quadratic methods of Chapter 2: heapsort, which hides a tree inside an array, and Quicksort, which divides an array by value around a pivot. Chapter 6 turned the tree from a hidden shape into a real data structure, the binary search tree, and then repaired its one great weakness — that it can grow as tall as a chain — with the self-balancing AVL tree. Chapter 7 went back to the recursion that powers all of these and asked when recursion wastes its effort, which led to memoization and dynamic programming.

This chapter does not introduce new theory. It does something just as important: it takes the theory of those three chapters and *uses* it, on problems large enough that several ideas must cooperate. Each problem is stated, then analyzed until its real content is visible, then solved step by step with every intermediate value written down, then implemented as a complete program whose printed output appears in the text exactly as the machine produced it. Most problems end with the wrong turns that readers most often take on them, and the new problems of this chapter end with a *check yourself* variant — a second instance of the same problem, with its answer, so that you can test whether you can now do it alone.

Table 8.1 lists every numbered problem of the chapter, where it is solved, and which earlier chapter supplies 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 problems draw on more than their "home" chapter. A problem about trees still needs a loop invariant from Chapter 2 to prove it correct and a recurrence from Chapter 3 to price it. **Table 8.2** is the map: what each earlier chapter contributes, and where in this chapter you will see it used.

Table 8.2 The revision map. Every chapter before this one contributes something the problems of this chapter actually need.

Ch	What it contributes	Where it is used here
1	The five-phase framework: understand, abstract, design, analyze, implement	§8.2, the order in which to write
1	The two goals: correctness first, then efficiency	§8.6.3, the local check is fast and wrong
2	The RAM model: which operations cost one unit	§8.8.4, why "one path" is the whole answer
2	Loop invariants and their three obligations	§8.4.3, the four regions of the partition; §8.6.4, the bounds invariant
2	Binary search and the $\Theta(\lg n)$ descent	§8.8.3, the rightmost path is a descent
3	O, Ω, Θ and how to state a bound honestly	§8.8.5, $O(h)$ versus $O(\lg n)$
3	Summations: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$	§8.4.5, Quicksort's worst case; §8.9.3, recomputing heights
3	Recursion trees and the Master Theorem	§8.4.6, Quicksort's best case; §8.9.6, the two balance checks
5	Heaps, `MAX-HEAPIFY`, `BUILD-MAX-HEAP`, heapsort	Problem 8.1; §8.6.6, why the heap property is not the BST property
5	Quicksort's partition: divide by value, not by position	Problem 8.2; §8.6.4, the bounds are a partition of the key space
6	The BST property, traversals, minimum and maximum	Problems 8.3, 8.4, 8.5
6	Balance factors, rotations, the four cases, $1.44\lg n$	Problems 8.6 to 8.10, and 8.11 to 8.15
7	Recursion that returns a *value* upward, not just a side effect	§8.9.4, the sentinel technique
7	Memoization and dynamic programming	Problems 8.16 and 8.17; §8.9.3, the naive height recomputation

THE ONE IDEA BEHIND ALMOST EVERY PROBLEM HERE. Nearly every problem in this chapter is the same idea in different clothes: *carry information down the recursion, or return it up, so that no piece of work is ever done twice.* `MAX-HEAPIFY` carries one misplaced value down until it settles. Problem 8.3 carries a pair of bounds down. Problem 8.6 returns heights up. Problems 8.7 to 8.10 return rebuilt subtrees up. The dynamic programs of Problems 8.16 and 8.17 store every subproblem's answer so that it is computed once and then only looked up. Problems 8.4 and 8.5 carry nothing and return nothing — and they are the two simplest problems in the chapter. If you understand *down-and-up information flow in a recursion*, you can derive every answer in this chapter from scratch.

8.2 How to Approach a Multi-Part Problem

Several problems in this chapter come in families: Problems 8.7 to 8.10 are four routines that only work together, and Problems 8.18 to 8.25 are eight routines over one data structure. Large problems like these reward a little method before any code is written. None of what follows is specific to trees; it is the five-phase framework of Chapter 1, compressed to fit a single problem.

Read every part before solving any part, and draw the dependencies. Reading Problems 8.7 to 8.10 together reveals three facts that are invisible when you read them one at a time: insertion (Problem 8.7) and deletion (Problem 8.10) both *call* the two rebalancers; the second rebalancer (Problem 8.9) is the mirror image of the first (Problem 8.8); and deletion needs a maximum, which Problem 8.5 has already written. So the sensible order of work is 8.8, then 8.9 (a transcription of 8.8), then 8.7, then 8.10 — not the order in which the problems are numbered. Solving in dependency order means each routine is written once.

Before writing code, write five things, in this order.

1. **The signature, exactly.** `bool isBST(MyTree t)`. The signature fixes what the routine receives and what it must hand back, and a surprising number of wrong solutions are correct answers to a slightly different signature.
2. **The base case, in words.** "The empty tree is a BST." "The empty tree has height -1." "`max` of an empty tree is undefined." Every routine in this chapter is a recursion, every recursion needs a base case, and naming it in words before coding it prevents the most common crash of all: dereferencing the empty tree.
3. **The invariant or the contract, in one sentence.** "Every value in this subtree lies strictly between `lowerBound` and `upperBound`." "The left subtree may have grown taller by one notch; both subtrees are already balanced." This sentence *is* the design. Write it as a comment directly above the function.
4. **The smallest failing example.** Not a big input — the *smallest* input on which a naive solution gives the wrong result. For Problem 8.3 that is the three-node chain of §8.6.3; for Problem 8.8 it is the three-node zig-zag of §8.11.4. Writing it down forces you to check your solution against it.
5. **The complexity claim, with its justification in half a sentence.** "$\Theta(n)$: each node is visited once." "$O(h)$: one recursive call per level." Do not write "$O(\lg n)$" unless you can say why the tree is short; §8.8.5 explains why that distinction matters.

In practice these five steps take three minutes and save far more.

Test the tiny cases first. The empty input, a single element, two elements and an absent key find more bugs than a fifteen-element example does, and they take seconds. Every program in this chapter tests them.

Say what is not yet handled. If a routine is unfinished — say, a deletion that does not yet cover a node with two children — write that down next to the code, with the input on which it fails. A known gap can be closed tomorrow; an unnoticed one ships.

> **THE MIRROR PRINCIPLE, AS A WORK-SAVER.** Reflecting a binary tree left-to-right exchanges left and right, $<$ and $>$, minimum and maximum, and the two rotations. Any correct routine, reflected symbol by symbol, is a correct routine for the mirrored problem. Spotting a mirror pair before starting — `max` and `min`, the two rebalancers, the two rotations of a red-black tree — halves the work. §8.12.2 states the principle precisely.

8.3 Problem 8.1 — Building a Max-Heap, Step by Step

8.3.1 The problem

Take the ten-element array

$$A = [5,\ 13,\ 2,\ 25,\ 7,\ 17,\ 20,\ 8,\ 4,\ 11].$$

1. Draw A as the nearly complete binary tree it represents, with each node's index.
2. Run `BUILD-MAX-HEAP` on A. After every call of `MAX-HEAPIFY`, write down the whole array, and list every swap the call makes.
3. Count the key comparisons and the swaps the construction makes in total.
4. Sort the resulting heap with heapsort, and give the array after each extraction of the maximum.
5. Explain why `BUILD-MAX-HEAP` costs $O(n)$ and not $O(n\lg n)$, and check the explanation against a measurement.

8.3.2 What you need from Chapter 5

A **max-heap** is an array $A[1..n]$ that satisfies the **max-heap property**: for every index $i > 1$, $A[\text{PARENT}(i)] \geq A[i]$. The tree is never stored; it is computed from the indices:

$$\text{PARENT}(i) = \lfloor i/2 \rfloor, \qquad \text{LEFT}(i) = 2i, \qquad \text{RIGHT}(i) = 2i + 1.$$

For example, the children of index 3 are indices 6 and 7, and the parent of index 10 is index 5.

The engine is `MAX-HEAPIFY`, **Pseudocode 8.1**. Its precondition is that the two subtrees of node i are already max-heaps, while $A[i]$ itself may be too small. It lets that one value *sink*: it compares the node with its children, swaps it with the larger child if that child is bigger, and repeats one level down.

Pseudocode 8.1 `MAX-HEAPIFY`: the value at node i sinks until neither child is larger than it.

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

`BUILD-MAX-HEAP`, **Pseudocode 8.2**, calls `MAX-HEAPIFY` on every internal node, from the last one back to the root. The nodes $\lfloor n/2 \rfloor + 1, \ldots, n$ are leaves, and a leaf is already a one-element heap, so the loop starts at $\lfloor n/2 \rfloor$. It counts *downward* so that both children of a node are finished before the node itself is processed — which is exactly the precondition `MAX-HEAPIFY` needs.

Pseudocode 8.2 `BUILD-MAX-HEAP`: repair the internal nodes from the bottom up.

```
BUILD-MAX-HEAP(A, n)
A.heap-size = n
for i = ⌊n/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

8.3.3 Step 1: the array as a tree

Here $n = 10$, so the internal nodes are $1, \ldots, \lfloor 10/2 \rfloor = 5$ and the leaves are $6, \ldots, 10$. Placing $A[1]$ at the root, $A[2]$ and $A[3]$ on the next level, $A[4]$ to $A[7]$ on the next, and $A[8]$ to $A[10]$ on the last, from left to right, gives the tree below. Each node is written as *value* with its index in brackets.

```
                    5 [1]
              /                 \
         13 [2]                  2 [3]
        /      \                /      \
    25 [4]     7 [5]        17 [6]     20 [7]
    /    \       /
 8 [8]  4 [9]  11 [10]
```

Check the index arithmetic on two nodes: node 4 has children $2 \cdot 4 = 8$ and $2 \cdot 4 + 1 = 9$, holding 8 and 4; node 5 has children 10 and 11, but $11 > n$, so node 5 has only a left child, holding 11. The tree is badly out of order — the root holds 5 while 25 sits two levels down.

8.3.4 Step 2: the trace, one call at a time

The loop runs $i = 5,4,3,2,1$. At each call we write down every comparison of line 3 and line 6, because those are the comparisons we are asked to count.

Call `MAX-HEAPIFY(A, 5)`. Node 5 holds 7. Its left child is index 10, holding 11; its right child would be index 11, which does not exist.

- Line 3: $10 \leq 10$ and $A[10] = 11 > A[5] = 7$? **Yes**, so $largest = 10$. (1 comparison.)
- Line 6: $r = 11 > 10$, so the right child is skipped; no comparison.
- Line 8: $largest = 10 \neq 5$, so **swap** $A[5] = 7$ with $A[10] = 11$.
- The recursive call `MAX-HEAPIFY(A, 10)` finds $\text{LEFT}(10) = 20 > 10$: node 10 is a leaf, nothing is compared, and the call ends.

$A = [5,13,2,25,\mathbf{11},17,20,8,4,\mathbf{7}]$. One comparison, one swap.

Call `MAX-HEAPIFY(A, 4)`. Node 4 holds 25, with children $A[8] = 8$ and $A[9] = 4$.

- Line 3: $8 > 25$? No, so $largest = 4$. (1 comparison.)
- Line 6: $4 > A[largest] = 25$? No. (1 comparison.)
- $largest = i$: nothing to do.

A is unchanged. Two comparisons, no swap. The value 25 is already larger than everything below it.

Call `MAX-HEAPIFY(A, 3)`. Node 3 holds 2, with children $A[6] = 17$ and $A[7] = 20$.

- Line 3: $17 > 2$? **Yes**, $largest = 6$. (1 comparison.)
- Line 6: $20 > A[6] = 17$? **Yes**, $largest = 7$. (1 comparison.) Note that the right child is compared with the *current champion* 17, not with the original 2.
- **Swap** $A[3] = 2$ with $A[7] = 20$. The call `MAX-HEAPIFY(A, 7)` finds that node 7 is a leaf ($\text{LEFT}(7) = 14 > 10$) and stops.

$A = [5,13,\mathbf{20},25,11,17,\mathbf{2},8,4,7]$. Two comparisons, one swap.

Call `MAX-HEAPIFY(A, 2)`. Node 2 holds 13, with children $A[4] = 25$ and $A[5] = 11$.

- Line 3: $25 > 13$? **Yes**, $largest = 4$. Line 6: $11 > 25$? No. (2 comparisons.)
- **Swap** $A[2] = 13$ with $A[4] = 25$.
- Recursive call `MAX-HEAPIFY(A, 4)`: node 4 now holds 13, with children $A[8] = 8$ and $A[9] = 4$. $8 > 13$? No. $4 > 13$? No. (2 comparisons.) Stop.

$A = [5,\mathbf{25},20,\mathbf{13},11,17,2,8,4,7]$. Four comparisons, one swap.

Call `MAX-HEAPIFY(A, 1)`. The root holds 5, the smallest value on the upper levels, and it will sink all the way down.

- At node 1: children $A[2] = 25$ and $A[3] = 20$. $25 > 5$? **Yes**, $largest = 2$. $20 > 25$? No. (2 comparisons.) **Swap** $A[1] \leftrightarrow A[2]$: now $A[1] = 25$, $A[2] = 5$.
- At node 2: children $A[4] = 13$ and $A[5] = 11$. $13 > 5$? **Yes**. $11 > 13$? No. (2 comparisons.) **Swap** $A[2] \leftrightarrow A[4]$: now $A[2] = 13$, $A[4] = 5$.
- At node 4: children $A[8] = 8$ and $A[9] = 4$. $8 > 5$? **Yes**. $4 > 8$? No. (2 comparisons.) **Swap** $A[4] \leftrightarrow A[8]$: now $A[4] = 8$, $A[8] = 5$.
- At node 8: $\text{LEFT}(8) = 16 > 10$, a leaf. Stop.

$A = [\mathbf{25},\mathbf{13},20,\mathbf{8},11,17,2,\mathbf{5},4,7]$. Six comparisons, three swaps.

Table 8.3 collects the five calls.

Table 8.3 `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**	

Check the result. A max-heap claim is checked pair by pair, parent against each child:

- node 1 (25): children 13 and 20 — both smaller ✓
- node 2 (13): children 8 and 11 ✓
- node 3 (20): children 17 and 2 ✓
- node 4 (8): children 5 and 4 ✓
- node 5 (11): child 7 ✓

Every internal node is at least as large as its children, so [25,13,20,8,11,17,2,5,4,7] is a max-heap, and its maximum 25 is at the root, as it must be.

8.3.5 Step 3: heapsort on the result

Heapsort repeats three moves: exchange $A[1]$, the maximum, with the last element of the heap; shrink the heap by one, so that the maximum is now outside it and in its final place; and call `MAX-HEAPIFY(A, 1)` to repair the root. Work the first three extractions by hand.

First extraction. Exchange $A[1] = 25$ with $A[10] = 7$, and shrink the heap to 9 elements: [7,13,20,8,11,17,2,5,4 | 25]. Now sink 7: its children are 13 and 20; the larger is 20 at index 3; swap, giving $A[1] = 20$, $A[3] = 7$. At index 3 the children are $A[6] = 17$ and $A[7] = 2$; $17 > 7$, so swap, giving $A[3] = 17$, $A[6] = 7$. Index 6 has $\text{LEFT}(6) = 12 > 9$, a leaf. Result: [20,13,17,8,11,7,2,5,4 | 25].

Second extraction. Exchange $A[1] = 20$ with $A[9] = 4$; heap size 8: [4,13,17,8,11,7,2,5 | 20,25]. Sink 4: children 13 and 17, larger 17 at index 3; swap. At index 3: children $A[6] = 7$ and $A[7] = 2$; $7 > 4$; swap. Result: [17,13,7,8,11,4,2,5 | 20,25].

Third extraction. Exchange $A[1] = 17$ with $A[8] = 5$; heap size 7: [5,13,7,8,11,4,2 | 17,20,25]. Sink 5: children 13 and 7, larger 13 at index 2; swap. At index 2: children $A[4] = 8$ and $A[5] = 11$; larger 11; $11 > 5$; swap. Index 5 has $\text{LEFT}(5) = 10 > 7$. Result: [13,11,7,8,5,4,2 | 17,20,25].

The remaining six extractions follow the same pattern, and the program in §8.3.7 prints every one of them. After the ninth, the heap holds one element, which is necessarily the smallest, and the array reads [2,4,5,7,8,11,13,17,20,25]: sorted, in place, with no second array.

8.3.6 Step 4: why the construction costs only $O(n)$

A quick estimate says: $\lfloor n/2 \rfloor$ calls, each costing $O(\lg n)$, so $O(n \lg n)$. That bound is true but not tight, and the reason it is loose is visible in **Table 8.3**: most calls are made on nodes near the bottom, where there is almost nowhere to sink. The call at node 5 could sink at most one level; only the call at the root could sink three.

Make that precise. A node of **height** h — the number of edges on the longest downward path from it to a leaf — costs $O(h)$ in `MAX-HEAPIFY`, because the value can sink at most h levels. A heap of n elements has at most $\lceil n/2^{h+1} \rceil$ nodes of height h (Chapter 5 proves this). Summing over all heights,

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(2n) = O(n), \tag{8.1}$$

using the series $\sum_{h \geq 0} h/2^h = 2$ from Chapter 3.

For a **perfect** heap we can even find the exact worst case. Let $n = 2^k - 1$, so the tree has levels $0, 1, \ldots, k-1$, with 2^ℓ nodes on level ℓ, each of height $k - 1 - \ell$. The total number of levels all the values could sink is the sum of the heights:

$$\sum_{\ell=0}^{k-1} 2^\ell (k - 1 - \ell) = 2^k - k - 1 = n - \lg(n+1). \tag{8.2}$$

Check it for $k = 3$, $n = 7$: the root has height 2, its two children height 1, the four leaves height 0, so the sum is $2 + 1 + 1 = 4$, and $7 - \lg 8 = 7 - 3 = 4$. ✓

Each level sunk is one swap and at most two comparisons, so by Equation (8.2) `BUILD-MAX-HEAP` never makes more than $n - \lg(n+1)$ swaps on a perfect heap — fewer than n, whatever the input. Equation (8.1) is the general statement of the same fact. The program below measures the swaps on keys in *ascending* order, where every value turns out to sink as far as it possibly can; the measured counts equal $n - \lg(n+1)$ exactly, from $n = 7$ up to $n = 32\,767$.

8.3.7 The program

Listing 8.1 follows **Pseudocode 8.1** and **Pseudocode 8.2** line for line, with a 1-based array so that the index arithmetic is exactly $2i$ and $2i + 1$. It prints the trace of §8.3.4, the check-yourself array of §8.3.8, the heapsort of §8.3.5 and the measurement of §8.3.6.

Listing 8.1 Problem 8.1 in full: `MAX-HEAPIFY` and `BUILD-MAX-HEAP` with a call-by-call trace, heapsort with the array after every extraction, and a measurement of the linear bound.

```
// buildheap.cpp -- Problem 8.1: BUILD-MAX-HEAP traced call by call, then heapsort.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <utility>
using namespace std;

vector<int> A;            // A[0] is unused, so the heap occupies A[1..n] as in the
pseudocode
int heapSize = 0;
long long comparisons = 0, swaps = 0;

int LEFT(int i)  { return 2 * i; }
int RIGHT(int i) { return 2 * i + 1; }

string show() {                          // the whole array, heap part and sorted tail
alike
    string s = "[";
    for (size_t k = 1; k < A.size(); ++k) s += (k > 1 ? " " : "") + to_string(A[k]);
    return s + "]";
}

// MAX-HEAPIFY exactly as in the pseudocode, with every swap reported when trace is on.
void maxHeapify(int i, bool trace) {
    int l = LEFT(i), r = RIGHT(i), largest = i;
    if (l <= heapSize) { ++comparisons; if (A[l] > A[i])       largest = l; }
    if (r <= heapSize) { ++comparisons; if (A[r] > A[largest]) largest = r; }
    if (largest != i) {
        if (trace) cout << "      swap A[" << i << "]=" << A[i] << " with A[" << largest
                        << "]=" << A[largest] << "\n";
        swap(A[i], A[largest]); ++swaps;
        maxHeapify(largest, trace);          // the value keeps sinking
    }
}

void buildMaxHeap(bool trace) {
    heapSize = (int)A.size() - 1;
    for (int i = heapSize / 2; i >= 1; --i) {
        if (trace) cout << "  MAX-HEAPIFY(A, " << i << ")  node value " << A[i] << "\n";
        maxHeapify(i, trace);
        if (trace) cout << "    A = " << show() << "\n";
    }
}
```

Listing 8.1 (Continued) *Problem 8.1 in full: MAX-HEAPIFY and BUILD-MAX-HEAP with a call-by-call trace, heapsort with the array after every extraction, and a measurement of the linear bound.*

```
bool isMaxHeap() {
    for (int i = 2; i <= heapSize; ++i) if (A[i / 2] < A[i]) return false;
    return true;
}

void heapsort(bool trace) {
    buildMaxHeap(false);
    for (int i = (int)A.size() - 1; i >= 2; --i) {
        swap(A[1], A[i]); ++swaps;           // the maximum goes to its final place
        --heapSize;
        maxHeapify(1, false);
        if (trace) cout << "  after extracting " << setw(2) << A[i] << ": " << show()
                        << "   heap-size = " << heapSize << "\n";
    }
}

void run(const vector<int>& input, const char* name) {
    A = input; A.insert(A.begin(), 0);
    comparisons = swaps = 0;
    cout << name << "\n  input A = " << show() << "\n";
    buildMaxHeap(true);
    cout << "  result is a max-heap: " << (isMaxHeap() ? "yes" : "NO")
         << "   comparisons = " << comparisons << ", swaps = " << swaps << "\n\n";
}

int main() {
    run({5, 13, 2, 25, 7, 17, 20, 8, 4, 11}, "BUILD-MAX-HEAP, the worked problem");
    run({3, 9, 2, 1, 4, 5}, "BUILD-MAX-HEAP, the check-yourself array");

    cout << "HEAPSORT on the worked array\n";
    A = {0, 5, 13, 2, 25, 7, 17, 20, 8, 4, 11};
    comparisons = swaps = 0;
    heapsort(true);
    cout << "  sorted: " << show() << "\n";

    cout << "\nThe linear bound: swaps made by BUILD-MAX-HEAP on n = 2^k - 1 keys in\n"
            "ascending order (every value must sink as far as it can)\n";
    cout << "      n     swaps    n - lg(n+1)\n";
    for (int k = 3; k <= 15; k += 3) {
        int n = (1 << k) - 1;
        A.assign(n + 1, 0);
        for (int v = 1; v <= n; ++v) A[v] = v;
        swaps = 0; buildMaxHeap(false);
        cout << setw(7) << n << setw(10) << swaps << setw(15) << (n - k) << "\n";
    }
    return 0;
```

Listing 8.1 (Continued) *Problem 8.1 in full: MAX-HEAPIFY and BUILD-MAX-HEAP with a call-by-call trace, heapsort with the array after every extraction, and a measurement of the linear bound.*

```
}
```

Compiled with `g++ -std=c++17 -O2 -Wall` and run, it prints:

```
BUILD-MAX-HEAP, the worked problem
  input A = [5 13 2 25 7 17 20 8 4 11]
  MAX-HEAPIFY(A, 5)  node value 7
      swap A[5]=7 with A[10]=11
    A = [5 13 2 25 11 17 20 8 4 7]
  MAX-HEAPIFY(A, 4)  node value 25
    A = [5 13 2 25 11 17 20 8 4 7]
  MAX-HEAPIFY(A, 3)  node value 2
      swap A[3]=2 with A[7]=20
    A = [5 13 20 25 11 17 2 8 4 7]
  MAX-HEAPIFY(A, 2)  node value 13
      swap A[2]=13 with A[4]=25
    A = [5 25 20 13 11 17 2 8 4 7]
  MAX-HEAPIFY(A, 1)  node value 5
      swap A[1]=5 with A[2]=25
      swap A[2]=5 with A[4]=13
      swap A[4]=5 with A[8]=8
    A = [25 13 20 8 11 17 2 5 4 7]
  result is a max-heap: yes   comparisons = 15, swaps = 6

BUILD-MAX-HEAP, the check-yourself array
  input A = [3 9 2 1 4 5]
  MAX-HEAPIFY(A, 3)  node value 2
      swap A[3]=2 with A[6]=5
    A = [3 9 5 1 4 2]
  MAX-HEAPIFY(A, 2)  node value 9
    A = [3 9 5 1 4 2]
  MAX-HEAPIFY(A, 1)  node value 3
      swap A[1]=3 with A[2]=9
      swap A[2]=3 with A[5]=4
    A = [9 4 5 1 3 2]
  result is a max-heap: yes   comparisons = 7, swaps = 3

HEAPSORT on the worked array
  after extracting 25: [20 13 17 8 11 7 2 5 4 25]   heap-size = 9
  after extracting 20: [17 13 7 8 11 4 2 5 20 25]   heap-size = 8
  after extracting 17: [13 11 7 8 5 4 2 17 20 25]   heap-size = 7
  after extracting 13: [11 8 7 2 5 4 13 17 20 25]   heap-size = 6
  after extracting 11: [8 5 7 2 4 11 13 17 20 25]   heap-size = 5
  after extracting  8: [7 5 4 2 8 11 13 17 20 25]   heap-size = 4
  after extracting  7: [5 2 4 7 8 11 13 17 20 25]   heap-size = 3
  after extracting  5: [4 2 5 7 8 11 13 17 20 25]   heap-size = 2
  after extracting  4: [2 4 5 7 8 11 13 17 20 25]   heap-size = 1
  sorted: [2 4 5 7 8 11 13 17 20 25]

The linear bound: swaps made by BUILD-MAX-HEAP on n = 2^k - 1 keys in
ascending order (every value must sink as far as it can)
      n     swaps    n - lg(n+1)
      7         4              4
     63        57             57
    511       502            502
   4095      4083           4083
  32767     32752          32752
```

Read the output against the hand work. The five blocks of the first trace are the five rows of **Table 8.3**, swap for swap, and the totals are 15 comparisons and 6 swaps. The first three heapsort lines are the three extractions of §8.3.5. The last table confirms the exact bound of §8.3.6 at five sizes: 32 767 keys need only 32 752 swaps to become a heap — fewer swaps than keys.

8.3.8 Check yourself

Run `BUILD-MAX-HEAP` by hand on $A = [3,9,2,1,4,5]$, counting comparisons and swaps, before reading the answer.

ANSWER. $n = 6$, so the loop runs $i = 3,2,1$. $i = 3$: node 3 holds 2, its only child is $A[6] = 5$; $5 > 2$, swap: [3,9,5,1,4,2] (1 comparison, 1 swap). $i = 2$: node 2 holds 9, children $A[4] = 1$ and $A[5] = 4$; neither is larger: no change (2 comparisons). $i = 1$: the root 3 has children 9 and 5; 9 is larger; swap: [9,3,5,1,4,2]. At node 2, 3 has children 1 and 4; 4 is larger; swap: [9,4,5,1,3,2]. Node 5 is a leaf (4 comparisons, 2 swaps). **Result** [9,4,5,1,3,2], with **7 comparisons and 3 swaps** in total — the second block of the output of **Listing 8.1**.

8.3.9 Common wrong turns on this problem

1. **Counting the loop upward**, `for i = 1 to ⌊n/2⌋`. The root is then heapified while its subtrees are still disordered, the precondition of `MAX-HEAPIFY` fails, and the result is not a heap. On [5,13,2,25,7,17,20,8,4,11] the upward loop leaves 25 below 13.
2. **Comparing the right child with $A[i]$ instead of $A[largest]$.** At node 3 of §8.3.4 this would compare 20 with 2 rather than with 17 — here the answer happens to agree, but when the left child is the larger one the wrong comparison picks the right child anyway, and the heap property fails one level down.
3. **Starting the loop at n instead of $\lfloor n/2 \rfloor$.** Correct, but half the calls are wasted on leaves.
4. **Mixing 0-based and 1-based indices.** In a C++ array that starts at index 0, the children of i are $2i + 1$ and $2i + 2$. **Listing 8.1** leaves $A[0]$ unused precisely so that the code matches the pseudocode.
5. **Forgetting to shrink the heap in heapsort.** If `heap-size` is not decremented, the next `MAX-HEAPIFY` pulls the extracted maximum straight back to the root.

8.4 Problem 8.2 — Quicksort's Partition, Traced; Best and Worst Cases

8.4.1 The problem

Take the nine-element array

$$A = [7,\ 2,\ 9,\ 4,\ 1,\ 8,\ 3,\ 6,\ 5].$$

1. Trace Lomuto's `PARTITION(A, 1, 9)`, one row per value of the loop index j: give $A[j]$, the decision, the new value of i, and the whole array after the step. Say where the pivot ends up.
2. Trace the complete `QUICKSORT(A, 1, 9)`: every call, its pivot, and the comparisons it makes. Give the total number of comparisons.
3. Which input of n distinct keys makes Quicksort (with this partition) as slow as possible? Count its comparisons exactly.
4. Which input makes it as fast as possible? Count its comparisons exactly when $n = 2^k - 1$.
5. Check both counts against a measurement.

8.4.2 What you need from Chapter 5

`PARTITION`, **Pseudocode 8.3**, takes the last element $x = A[r]$ as the **pivot** and rearranges $A[p..r]$ so that everything $\leq x$ comes first, then x, then everything $> x$. It returns the pivot's final index q.

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
```

The loop maintains the **four-region invariant** of Chapter 5: at the start of each iteration, $A[p..i]$ holds values $\leq x$, $A[i+1..j-1]$ holds values $>x$, $A[j..r-1]$ is not yet examined, and $A[r] = x$.

```
p              i         j                r
+--------------+---------+----------------+---+
|    <= x      |   > x   |   unexamined   | x |
+--------------+---------+----------------+---+
```

`QUICKSORT`, **Pseudocode 8.4**, partitions and then sorts the two sides independently. The pivot is already in its final place and is never touched again.

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)
```

8.4.3 Step 1: the partition, one step per line

Here $p = 1$, $r = 9$, the pivot is $x = A[9] = 5$, and i starts at $p - 1 = 0$, meaning "no value ≤ 5 found yet". The loop runs $j = 1, \ldots, 8$, and each iteration makes exactly one comparison, on line 4.

- $j = 1$: $A[1] = 7$. Is $7 \leq 5$? No. Nothing happens; 7 joins the "$>x$" region. $A = [7,2,9,4,1,8,3,6,5]$, $i = 0$.
- $j = 2$: $A[2] = 2 \leq 5$. Yes: i becomes 1, and $A[1] = 7$ is exchanged with $A[2] = 2$. $A = [\mathbf{2}, \mathbf{7}, 9,4,1,8,3,6,5]$. The small value has jumped over the large one.
- $j = 3$: $A[3] = 9 > 5$. No change; $i = 1$.
- $j = 4$: $A[4] = 4 \leq 5$. $i = 2$; exchange $A[2] = 7$ with $A[4] = 4$. $A = [2, \mathbf{4}, 9, \mathbf{7}, 1,8,3,6,5]$.
- $j = 5$: $A[5] = 1 \leq 5$. $i = 3$; exchange $A[3] = 9$ with $A[5] = 1$. $A = [2,4, \mathbf{1}, 7, \mathbf{9}, 8,3,6,5]$.
- $j = 6$: $A[6] = 8 > 5$. No change; $i = 3$.
- $j = 7$: $A[7] = 3 \leq 5$. $i = 4$; exchange $A[4] = 7$ with $A[7] = 3$. $A = [2,4,1, \mathbf{3}, 9,8, \mathbf{7}, 6,5]$.
- $j = 8$: $A[8] = 6 > 5$. No change; $i = 4$.

The loop is over. Line 7 exchanges $A[i+1] = A[5] = 9$ with the pivot $A[9] = 5$:

$$A = [2,\ 4,\ 1,\ 3,\ \mathbf{5},\ 8,\ 7,\ 6,\ 9], \qquad q = 5.$$

Table 8.4 records the same steps in one table.

Table 8.4 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]
line 7				$A[5] \leftrightarrow A[9]$	[2,4,1,3,5,8,7,6,9]

Check the invariant at the end. $A[1..4] = \{2,4,1,3\}$, all ≤ 5 ✓. $A[5] = 5$, the pivot ✓. $A[6..9] = \{8,7,6,9\}$, all >5 ✓. The pivot is in the place it will occupy in the sorted array — there are exactly four values smaller than 5. The partition made exactly $r - p = 8$ comparisons, one per loop iteration; that count does not depend on the data at all.

8.4.4 Step 2: the whole recursion

After the first partition, `QUICKSORT` sorts $A[1..4] = [2,4,1,3]$ and $A[6..9] = [8,7,6,9]$. Each call partitions its own range the same way.

- `QUICKSORT(A, 1, 4)` on [2,4,1,3], pivot 3. $j = 1$: $2 \le 3$, $i = 1$, exchange $A[1]$ with itself. $j = 2$: $4 > 3$. $j = 3$: $1 \le 3$, $i = 2$, exchange $A[2] = 4$ with $A[3] = 1$: [2,1,4,3]. Line 7: exchange $A[3] = 4$ with $A[4] = 3$: [2,1,3,4], $q = 3$. Three comparisons.
- `QUICKSORT(A, 1, 2)` on [2,1], pivot 1. $j = 1$: $2 > 1$. Line 7: exchange $A[1]$ with $A[2]$: [1,2], $q = 1$. One comparison. Both sides now have at most one element.
- `QUICKSORT(A, 6, 9)` on [8,7,6,9], pivot 9, the largest. Every $A[j] \le 9$, so i climbs with each j and each exchange is of an element with itself; line 7 exchanges $A[9]$ with itself. $q = 9$: the split is three elements against zero. Three comparisons.
- `QUICKSORT(A, 6, 8)` on [8,7,6], pivot 6, the smallest. $8 > 6$, $7 > 6$; line 7 exchanges $A[6] = 8$ with $A[8] = 6$: [6,7,8], $q = 6$. Two comparisons.
- `QUICKSORT(A, 7, 8)` on [7,8], pivot 8. $7 \le 8$; $q = 8$. One comparison.

Every other call has $p \ge r$ and returns at once without comparing anything. The recursion, drawn as a tree of ranges with each call's pivot, is:

```
                    [1..9] pivot 5
                 /                  \
        [1..4] pivot 3            [6..9] pivot 9
         /          \               /          \
  [1..2] pivot 1   [4..4]    [6..8] pivot 6   (empty)
                                /       \
                            (empty)   [7..8] pivot 8
```

Total comparisons: $8 + 3 + 1 + 3 + 2 + 1 = 18$. The right-hand branch shows what goes wrong in the worst case: the pivots 9 and then 6 were the largest and then the smallest of their ranges, so each partition peeled off only one element.

8.4.5 Step 3: the worst case

Suppose the pivot is always the **largest** value of its range. Then every partition of m elements makes $m-1$ comparisons and leaves one side with $m-1$ elements and the other side empty. Keys already in **ascending order** do exactly this with a last-element pivot: the last element of every range is its maximum. Writing $C(n)$ for the number of comparisons,

$$C(1) = 0, \qquad C(n) = C(n-1) + (n-1).$$

Unroll it: $C(n) = (n-1) + (n-2) + \cdots + 1 + 0 = \frac{n(n-1)}{2}$, the triangular sum of Chapter 3. In asymptotic form, with the constant-time empty call absorbed,

$$T(n) = T(n-1) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n^2). \tag{8.3}$$

For $n = 1023$ this is $\frac{1023 \times 1022}{2} = 522\,753$ comparisons. Keys in *descending* order are just as bad, because the pivot is then the smallest of every range, so the empty side is on the left instead. Equation (8.3) is the quadratic behavior that makes a naive Quicksort dangerous on data that arrive already sorted — which real data very often do.

8.4.6 Step 4: the best case

The best a partition can do is to put the pivot exactly in the middle, leaving $\frac{m-1}{2}$ elements on each side. Take $n = 2^k - 1$, so that every range has odd size and halves exactly. Then

$$C(1) = 0, \qquad C(n) = 2\,C\left(\frac{n-1}{2}\right) + (n-1).$$

In asymptotic form this is the merge-sort recurrence,

$$T(n) = 2T(n/2) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n\lg n), \tag{8.4}$$

by case 2 of the Master Theorem: $a = b = 2$, $n^{\log_b a} = n$, and $f(n) = \Theta(n)$.

The exact count. Compute the first values: $C(1) = 0$; $C(3) = 2 \cdot 0 + 2 = 2$; $C(7) = 2 \cdot 2 + 6 = 10$; $C(15) = 2 \cdot 10 + 14 = 34$. Guess $C(2^k - 1) = (k-2)\,2^k + 2$: for $k = 1,2,3,4$ this gives $0,2,10,34$ ✓. Prove it by induction on k. The base case $k = 1$ is $(1-2) \cdot 2 + 2 = 0$ ✓. For the step, assume it for k; then

$$\begin{aligned} C(2^{k+1} - 1) = 2[(k-2)2^k + 2] + (2^{k+1} - 2) &= (k-2)2^{k+1} + 2^{k+1} + 2 \\ &= ((k+1) - 2)2^{k+1} + 2, \end{aligned}$$

which is the formula for $k + 1$. ✓ Substituting $2^k = n + 1$ and $k = \lg(n+1)$,

$$C(n) = (n+1)\lg(n+1) - 2n. \tag{8.5}$$

For $n = 1023$: $1024 \times 10 - 2046 = 8\,194$ comparisons, against $522\,753$ in the worst case — sixty-four times fewer.

Which input achieves it? The pivot must be the median of every range it meets. **Listing 8.2** builds such an input recursively: put the median last; before it, the arrangement for the smaller

half, then the arrangement for the larger half. One subtlety is explained in the listing's comment — Lomuto's final exchange moves the first element of the "$>x$" region to the end, so the larger half is stored rotated by one place, and the partition rotates it back.

A **random** order lies between the two, and Chapter 5 showed that its expected count is about $2n\ln n \approx 1.39\, n\lg n$: far closer to the best case than to the worst. That is why Chapter 5 recommends choosing the pivot at random — it makes the expected cost $\Theta(n\lg n)$ on *every* input, including the sorted one.

8.4.7 The program

Listing 8.2 implements **Pseudocode 8.3** and **Pseudocode 8.4** with a comparison counter, prints the partition trace of §8.4.3 and the recursion of §8.4.4, partitions the check-yourself array of §8.4.8, and measures comparisons on sorted, best-case and random inputs of size $2^k - 1$.

Listing 8.2 Problem 8.2 in full: a traced Lomuto partition, the complete recursion with per-call comparison counts, a generator of best-case inputs, and a measurement of the best, worst and random cases.

```
// quicktrace.cpp -- Problem 8.2: Lomuto PARTITION traced step by step, the whole
// recursion of QUICKSORT, and the comparison counts of the best and worst cases.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <random>
#include <cmath>
#include <utility>
#include <algorithm>
using namespace std;

vector<int> A;                  // 1-based: A[0] is unused
long long comparisons = 0;
int maxDepth = 0;

string window(int p, int r) {  // A[p..r] as text
    string s = "[";
    for (int k = p; k <= r; ++k) s += (k > p ? " " : "") + to_string(A[k]);
    return s + "]";
}

// Lomuto's PARTITION, exactly as in the pseudocode. With trace on, one line per j.
int partition(int p, int r, bool trace) {
    int x = A[r];                                   // the pivot is the last element
    int i = p - 1;                                  // end of the "<= x" region
    for (int j = p; j <= r - 1; ++j) {
        ++comparisons;
        bool small = A[j] <= x;
        if (small) { ++i; swap(A[i], A[j]); }
        if (trace) {
            string action = small ? "<= x: i=" + to_string(i) + ", swap A[i],A[j]"
                                  : ">  x: no change";
            cout << "    j=" << j << "  A[j]=" << setw(2) << (small ? A[i] : A[j]) << "  "
                 << left << setw(28) << action << right << window(p, r) << "\n";
        }
    }
    swap(A[i + 1], A[r]);                           // drop the pivot into the gap
    if (trace) {
        string action = "final swap A[" + to_string(i + 1) + "] <-> A[" + to_string(r) +
"]";
        cout << "    " << left << setw(42) << action << right << window(p, r)
             << "   q = " << i + 1 << "\n";
    }
    return i + 1;
}
```

Listing 8.2 (Continued) *Problem 8.2 in full: a traced Lomuto partition, the complete recursion with per-call comparison counts, a generator of best-case inputs, and a measurement of the best, worst and random cases.*

```
void quicksort(int p, int r, int depth, bool trace) {
    if (p >= r) return;                             // 0 or 1 element: already sorted
    maxDepth = max(maxDepth, depth);
    int before = (int)comparisons;
    int q = partition(p, r, false);
    if (trace)
        cout << string(2 * depth, ' ') << "QUICKSORT(A," << p << "," << r << ")  pivot "
             << A[q] << " lands at q=" << q << "  (" << (int)comparisons - before
             << " comparison" << ((int)comparisons - before == 1 ? ")  " : "s)  ") <<
window(p, r) << "\n";
    quicksort(p, q - 1, depth + 1, trace);
    quicksort(q + 1, r, depth + 1, trace);
}

// Arrange lo..hi so that Lomuto's last-element pivot is always the median of its range.
// If the "<= x" values L come first and the "> x" values R next, PARTITION leaves L in
// place and its final swap moves R's first value to the end. So R is stored rotated
// one place to the right, and the partition itself rotates it back.
vector<int> bestCase(int lo, int hi) {
    if (lo > hi) return {};
    int mid = lo + (hi - lo) / 2;
    vector<int> L = bestCase(lo, mid - 1), R = bestCase(mid + 1, hi), out = L;
    if (!R.empty()) {
        out.push_back(R.back());                    // becomes R's last value again
        out.insert(out.end(), R.begin(), R.end() - 1);
    }
    out.push_back(mid);                             // the pivot, last
    return out;
}

long long sortCount(const vector<int>& v) {
    A = v; A.insert(A.begin(), 0);
    comparisons = 0; maxDepth = 0;
    quicksort(1, (int)v.size(), 0, false);
    return comparisons;
}

int main() {
    cout << "PARTITION(A, 1, 9) on [7 2 9 4 1 8 3 6 5], pivot x = A[9] = 5\n";
    A = {0, 7, 2, 9, 4, 1, 8, 3, 6, 5};
    comparisons = 0;
    partition(1, 9, true);
```

Listing 8.2 (Continued) *Problem 8.2 in full: a traced Lomuto partition, the complete recursion with per-call comparison counts, a generator of best-case inputs, and a measurement of the best, worst and random cases.*

```
    cout << "\nThe whole QUICKSORT on the same array\n";
    A = {0, 7, 2, 9, 4, 1, 8, 3, 6, 5};
    comparisons = 0; maxDepth = 0;
    quicksort(1, 9, 0, true);
    cout << "sorted: " << window(1, 9) << "   total comparisons = " << comparisons
         << ", deepest level = " << maxDepth << "\n";

    cout << "\nCheck yourself: PARTITION on [3 8 1 6 2 7 5 4], pivot 4\n";
    A = {0, 3, 8, 1, 6, 2, 7, 5, 4};
    partition(1, 8, true);

    cout << "\n     n   sorted   n(n-1)/2   best case   (n+1)lg(n+1)-2n   random\n";
    mt19937 rng(20260923);
    for (int n : {7, 15, 31, 63, 127, 255, 511, 1023}) {
        vector<int> sorted(n);
        for (int k = 0; k < n; ++k) sorted[k] = k + 1;
        long long w = sortCount(sorted);
        long long b = sortCount(bestCase(1, n));
        vector<int> rnd = sorted; shuffle(rnd.begin(), rnd.end(), rng);
        long long a = sortCount(rnd);
        long long k = llround(log2(n + 1.0));      // n = 2^k - 1
        cout << setw(6) << n << setw(9) << w << setw(11) << (long long)n * (n - 1) / 2
             << setw(12) << b << setw(18) << (n + 1) * k - 2LL * n << setw(9) << a << "\n";
    }
    return 0;
}
```

It prints:

```
PARTITION(A, 1, 9) on [7 2 9 4 1 8 3 6 5], pivot x = A[9] = 5
    j=1  A[j]= 7  >  x: no change             [7 2 9 4 1 8 3 6 5]
    j=2  A[j]= 2  <= x: i=1, swap A[i],A[j]   [2 7 9 4 1 8 3 6 5]
    j=3  A[j]= 9  >  x: no change             [2 7 9 4 1 8 3 6 5]
    j=4  A[j]= 4  <= x: i=2, swap A[i],A[j]   [2 4 9 7 1 8 3 6 5]
    j=5  A[j]= 1  <= x: i=3, swap A[i],A[j]   [2 4 1 7 9 8 3 6 5]
    j=6  A[j]= 8  >  x: no change             [2 4 1 7 9 8 3 6 5]
    j=7  A[j]= 3  <= x: i=4, swap A[i],A[j]   [2 4 1 3 9 8 7 6 5]
    j=8  A[j]= 6  >  x: no change             [2 4 1 3 9 8 7 6 5]
    final swap A[5] <-> A[9]                  [2 4 1 3 5 8 7 6 9]   q = 5

The whole QUICKSORT on the same array
QUICKSORT(A,1,9)  pivot 5 lands at q=5  (8 comparisons)  [2 4 1 3 5 8 7 6 9]
  QUICKSORT(A,1,4)  pivot 3 lands at q=3  (3 comparisons)  [2 1 3 4]
    QUICKSORT(A,1,2)  pivot 1 lands at q=1  (1 comparison)  [1 2]
  QUICKSORT(A,6,9)  pivot 9 lands at q=9  (3 comparisons)  [8 7 6 9]
    QUICKSORT(A,6,8)  pivot 6 lands at q=6  (2 comparisons)  [6 7 8]
      QUICKSORT(A,7,8)  pivot 8 lands at q=8  (1 comparison)  [7 8]
sorted: [1 2 3 4 5 6 7 8 9]   total comparisons = 18, deepest level = 3

Check yourself: PARTITION on [3 8 1 6 2 7 5 4], pivot 4
    j=1  A[j]= 3  <= x: i=1, swap A[i],A[j]   [3 8 1 6 2 7 5 4]
    j=2  A[j]= 8  >  x: no change             [3 8 1 6 2 7 5 4]
    j=3  A[j]= 1  <= x: i=2, swap A[i],A[j]   [3 1 8 6 2 7 5 4]
    j=4  A[j]= 6  >  x: no change             [3 1 8 6 2 7 5 4]
    j=5  A[j]= 2  <= x: i=3, swap A[i],A[j]   [3 1 2 6 8 7 5 4]
    j=6  A[j]= 7  >  x: no change             [3 1 2 6 8 7 5 4]
    j=7  A[j]= 5  >  x: no change             [3 1 2 6 8 7 5 4]
    final swap A[4] <-> A[8]                  [3 1 2 4 8 7 5 6]   q = 4

     n   sorted   n(n-1)/2   best case   (n+1)lg(n+1)-2n   random
     7       21         21          10                10       10
    15      105        105          34                34       45
    31      465        465          98                98      132
    63     1953       1953         258               258      333
   127     8001       8001         642               642      839
   255    32385      32385        1538              1538     2435
   511   130305     130305        3586              3586     4892
  1023   522753     522753        8194              8194    10637
```

The first block is **Table 8.4**, row for row. The second is the recursion tree of §8.4.4, with its 18 comparisons. The last table is the analysis made visible: the sorted column equals $\frac{n(n-1)}{2}$ at every size, as Equation (8.3) predicts, and the best-case column equals Equation (8.5) at every size, as §8.4.6 derived. The random column sits just above the best case, growing like $n\lg n$, while the sorted column explodes: at $n = 1023$ a sorted input costs fifty times what a shuffled one does.

8.4.8 Check yourself

Partition $A = [3,8,1,6,2,7,5,4]$ with `PARTITION(A, 1, 8)`. Give q, the array it leaves, and the number of comparisons.

> **ANSWER.** The pivot is $x = 4$ and i starts at 0. $j = 1$: $3 \leq 4$, $i = 1$, $A[1]$ is exchanged with itself. $j = 2$: $8 > 4$. $j = 3$: $1 \leq 4$, $i = 2$, exchange $A[2] = 8$ with $A[3] = 1$: $[3,1,8,6,2,7,5,4]$. $j = 4$: $6 > 4$. $j = 5$: $2 \leq 4$, $i = 3$, exchange $A[3] = 8$ with $A[5] = 2$: $[3,1,2,6,8,7,5,4]$. $j = 6,7$: $7 > 4$, $5 > 4$. Line 7: exchange $A[4] = 6$ with $A[8] = 4$. **Result** $[3,1,2,4,8,7,5,6]$ with $q = 4$, after 7 comparisons — the third block of the output of **Listing 8.2**. Three values are smaller than 4, so 4 belongs at index 4.

8.4.9 Common wrong turns on this problem

1. **Starting i at p instead of $p-1$.** The "$\leq x$" region then claims an element it has never examined, and the first small value overwrites it.
2. **Running j up to r instead of $r-1$.** The pivot compares with itself, joins the "$\leq x$" region, and line 7 then exchanges it with the wrong element.
3. **Recursing on $A[p..q]$ instead of $A[p..q-1]$.** When the pivot is the largest value, $q = r$, and the call `QUICKSORT(A, p, q)` is the same call again: infinite recursion.
4. **Saying the worst case is rare.** For a last-element pivot the worst case is *sorted input*, which is common. The honest statement is: deterministic Quicksort is $\Theta(n^2)$ in the worst case, and randomizing the pivot makes $\Theta(n\lg n)$ the *expected* cost on every input.
5. **Mixing Lomuto's and Hoare's return values.** Lomuto returns the pivot's final index, and the recursion excludes it. Hoare's scheme, also in Chapter 5, returns a split point j with the pivot not necessarily at j, and the recursion is on $A[p..j]$ and $A[j+1..r]$.

8.5 The Data Type the Tree Problems Use

Problems 8.3 to 8.10 all take a `MyTree`. Before we can solve any of them we must be exact about what a `MyTree` is, because every one of the eight solutions is a recursion over its structure, and a recursion over a structure you have not defined is guesswork.

We state the type precisely, and then we build it. A `MyTree` is one of exactly two things:

- the **empty tree**, which holds nothing; or
- a **node**, which holds one integer and two `MyTree`s, called the left and right subtrees.

That is the whole definition, and its shape is the shape of every solution: *if the tree is empty, do the easy thing; otherwise, do something with the integer and recurse on the two subtrees.*

> **DEFINITION (the `MyTree` type).** A `MyTree` is either empty, or a triple (v, L, R) where v is an integer and L and R are `MyTree`s. We write the empty tree as $\varnothing$. The **height** of a `MyTree` is

$$h(\varnothing) = -1, \qquad h((v, L, R)) = 1 + \max(h(L), h(R)). \tag{8.6}$$

> The **balance factor** of a non-empty tree is

$$\mathrm{bf}((v, L, R)) = h(R) - h(L), \tag{8.7}$$

> exactly the convention of Chapter 6: in Equation (8.7), negative means left-heavy, positive means right-heavy, zero means level.

Why is the empty tree's height -1 and not 0? Because then the formula of Equation (8.6) works for a leaf with no special case: a leaf has two empty subtrees, so $h = 1 + \max(-1, -1) = 1 + (-1) = 0$. A leaf has height 0; it has no edges below it. Some presentations — including several of the figures reproduced in Chapter 6 and in this chapter — count *nodes on the longest path* instead, which makes a leaf height 1 and the empty tree height 0 — one greater than ours everywhere. Both conventions are correct and both appear in the literature; what matters is that the balance factors, which are *differences* of heights, come out identically. We use the edge-counting convention throughout, as Chapter 6 did.

Listing 8.3 is the type, the operations, and a printer, as a program you can run.

Listing 8.3 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.

```
// 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 8.3 (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;
}
```

Compiled and run, it prints:

```
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
```

Learn to read that one-line shape notation, because every result in this chapter is reported in it. `13(-1){…,…}` means *the node* 13, *whose balance factor is* −1, *with the left subtree before the comma and the right subtree after*. A leaf is written without braces: `4(0)`. The empty tree is a full stop. So the string above says: root 13, left child 10 with children 5 and 11, node 5 with children 4 and 6, right child 15 with no left child and a right child 16. That is the eight-node tree on which Chapter 6 worked its insertion examples, and it is the tree we will keep returning to.

> **ONE CONSTRUCTOR, NO STALE HEIGHTS.** Notice that `node()` is the *only* way to build a tree, and that it recomputes the height every time. Chapter 6 §6.16.1 warned that the single most common bug in a hand-written AVL tree is refreshing the heights in the wrong order after a rotation. In the style of **Listing 8.3** that bug is **impossible**, because a rotation is expressed as a fresh `node(...)` built out of existing subtrees, and `node` always computes a parent's height after its children exist. This is the deep reason `insertHB` and `rebalanceForLeft` in Problems 8.7 and 8.8 *return* a `MyTree` instead of mutating one in place. Those signatures are chosen that way on purpose.

8.6 Problem 8.3 — `isBST`: Is This Tree a Binary Search Tree?

8.6.1 The problem

Write a function `bool isBST(MyTree t)` that decides whether the binary tree `t` is a binary search tree (BST). The function must be recursive, and it may delegate the work to a recursive helper that receives, besides the tree, a lower bound and an upper bound:

`isBST(t, lowerBound, upperBound)` decides whether `t` is a BST *all of whose values lie strictly between* `lowerBound` *and* `upperBound`.

Those two extra parameters are a very large hint, and §8.6.4 shows why.

8.6.2 What the problem is really about

Three things, in order of importance.

First: do you know what the BST property actually says? **Figure 8.1** states it: the left branch*es* contain the nodes whose values are less than the value of the node, and the right branch*es* contain the nodes whose values are greater. The plural is the whole question. It is not *the left child*; it is *every node anywhere in the left subtree*.

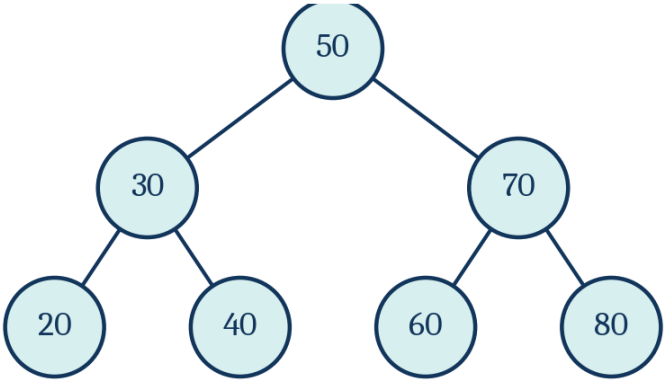


Figure 8.1 A binary search tree, illustrating the definition: "Left branch(es) contains the nodes whose values $<$ the value of the node" — branches, plural, not children.

Second: can you turn a global property into a local one? The property as stated quantifies over a whole subtree, and a recursion can only look at one node at a time. The art is finding a piece of information which, if carried down to each node, makes the global property checkable locally. That piece of information is the pair of bounds, and the helper's signature hands it to you.

Third: can you pick the sentinel values for the initial call? That matters less than the other two, but it is where the most tempting finished-looking solution goes wrong, and §8.6.7 works it out.

8.6.3 The wrong answer, and the smallest tree that exposes it

Almost everyone's first instinct is the *local* check: at each node, compare it with 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);
}
```

It is short, it is fast, it runs in $\Theta(n)$, and it is wrong. Here is the smallest tree that proves it. Take

20 at the root, 10 on the left with children 5 and 25, 30 on the right.

Check every parent–child pair by hand:

- At 20: left child is 10, and $10 < 20$. ✓ Right child is 30, and $30 > 20$. ✓
- At 10: left child is 5, and $5 < 10$. ✓ Right child is 25, and $25 > 10$. ✓
- At 5: a leaf. Nothing to check. ✓
- At 25: a leaf. ✓

- At 30: a leaf. ✓

Every local test passes. But the tree is **not** a BST, because 25 sits in the left subtree of 20 and $25 > 20$. The inorder walk is 5,10,25,20,30 — not increasing.

> **THIS IS THE TRAP.** A local parent–child check is a *necessary* condition for a BST and not a *sufficient* one. It is the first solution nearly everyone writes, and it passes every test built from small, well-behaved trees. Keep the four-node counterexample above in mind, together with the sentence "the local check is necessary but not sufficient": between them they are the whole content of the problem.

Four nodes is the minimum. Can three nodes do it? With three nodes you have a root and two others. If both hang off the root as children, the local check *is* the whole property. If they form a chain root–child–grandchild, the local check misses the root-versus-grandchild comparison — for example root 20, left child 10, and 10's right child 25: local checks pass at 20 ($10 < 20$) and at 10 ($25 > 10$), yet $25 > 20$ sits on the left. So **three nodes suffice** — root 20, left child 10, right child of 10 equal to 25 — and our four-node example is simply the more memorable one because it also has a leaf on each side. Exercise 2 asks you to confirm that two nodes can never expose the bug.

8.6.4 Deriving the right answer

We want a statement about a single node that, applied at every node, is equivalent to the global BST property. Here is the derivation, one step per line.

Step 1 — write the property we want to hold at the whole tree. For a tree t rooted at value v with subtrees L and R:

$$\mathrm{BST}(t) \Leftrightarrow (\forall u \in L: u < v) \wedge (\forall u \in R: u > v) \wedge \mathrm{BST}(L) \wedge \mathrm{BST}(R). \tag{8.8}$$

Step 2 — notice that the two universal quantifiers of Equation (8.8) **are range constraints.** "Every value in L is less than v" says: every value in L lies in the range $(-\infty, v)$. "Every value in R is greater than v" says: every value in R lies in $(v, +\infty)$.

Step 3 — generalize the claim so that it composes. Instead of asking "is t a BST?", ask the stronger question:

$P(t, \ell, u)$**:** *t is a BST and every value in t lies strictly between ℓ and u.*

Step 4 — expand P using step 1. Suppose $t = (v, L, R)$. Then $P(t, \ell, u)$ holds exactly when

1. $\ell < v < u$ — the root itself is inside the range; and
2. $P(L, \ell, v)$ — the left subtree is a BST whose values lie between ℓ and v; and
3. $P(R, v, u)$ — the right subtree is a BST whose values lie between v and u.

Check that this is right. Condition 2 forces every value of L below v, which is the first quantifier of step 1, *and* keeps them above ℓ, which is what the caller needs. Condition 3 does the mirror. Conditions 2 and 3 also recursively force $\mathrm{BST}(L)$ and $\mathrm{BST}(R)$. So P expands into exactly the three parts of the definition, with the ranges threaded through.

Step 5 — the base case. $P(\varnothing, \ell, u)$ is true for every ℓ and u: the empty tree is a BST, and the claim "every value in it lies between ℓ and u" is vacuously true because there are no values.

Step 6 — recover the original question. $\mathrm{BST}(t) \Leftrightarrow P(t, -\infty, +\infty)$, because with infinite bounds condition 1 is automatic.

That is the algorithm, and the recursion it describes is precisely the helper the problem suggests.

WHY THE BOUNDS WORK: A PARTITION OF THE KEY SPACE. Notice what the recursion is doing. At the root, the interval $(-\infty, +\infty)$ is cut at v into $(-\infty, v)$ and $(v, +\infty)$; at the next level each half is cut again. Every node owns a disjoint open interval of the key space, and the BST property is exactly the statement *every node's value lies inside the interval it owns*. This is the same move as Quicksort's partition in Chapter 5 and Problem 8.2: divide by **value**, not by position. Seeing that connection is seeing that Chapters 5 and 6 share one idea.

8.6.5 The program

Listing 8.4 contains the right answer, the wrong answer for contrast, an alternative answer by inorder walk, and a diagnostic version that names the first node to break its bound.

Listing 8.4 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.

```
// 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 8.4 (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 8.4 (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;
}
```

The program prints:

```
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
```

Line 2 of the second block is the whole lesson: `isBSTLocal` says **true** and `isBST` says **false**, on a four-node tree, and the diagnostic names the guilty node.

Table 8.5 traces the correct recursion on the trap tree by hand, so you can reproduce it with pencil and paper. Read it as a stack: each row is one call, with the bounds it received.

Table 8.5 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	$-\infty$	$+\infty$	$-\infty < 20 < +\infty$ ✓	recurse
2	10	$-\infty$	20	$-\infty < 10 < 20$ ✓	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

Notice row 4: the node 25 is rejected by the upper bound 20, which came from the *grandparent*. That is the mechanism. The bound is how a fact about an ancestor travels down to a descendant.

8.6.6 Why this is not the heap property

A short but valuable aside, because the two properties are so often confused. Chapter 5 built heaps, which also store numbers in a binary tree and also have a parent–child ordering rule. The rules are not the same, and the difference is exactly the difference between *local* and *global*.

- **The max-heap property** is $A[\text{parent}(i)] \geq A[i]$ — a rule between a node and its *children only*. It says nothing about left versus right. It is genuinely local, which is why the local check *is* the correct heap check.
- **The BST property** is a rule between a node and its whole *subtrees*. It is global, which is why the local check is wrong.

So: a heap can be validated by comparing each node with its two children, and a BST cannot. If someone asks "why does `isBST` need bounds when `isMaxHeap` does not?", that is the answer.

8.6.7 A tempting but wrong solution, and why it fails

Here is a version that looks finished. It is the bounded-interval idea exactly, with `int` bounds seeded with the smallest and largest values an `int` can hold:

```
bool isBSTNarrow(MyTree t, int lowerBound, int upperBound) {
    if (isEmpty(t)) return true;
    if (t->value <= lowerBound || t->value >= upperBound) return false;
    return isBSTNarrow(t->left,  lowerBound, t->value)
        && isBSTNarrow(t->right, t->value,   upperBound);
}
bool isBSTNarrow(MyTree t) { return isBSTNarrow(t, INT_MIN, INT_MAX); }
```

Compare it with `isBST` in **Listing 8.4**. The structure is identical, line for line: the base case, the strict double test, the two recursive calls with the node's own value substituted as the new bound on the appropriate side. **The derivation of §8.6.4 leads straight to it**, and that is exactly what makes it tempting. The strict comparisons `<=` and `>=` match the definition of **Figure 8.1**, which allows no two nodes with the same value; duplicates are rejected, as the `dup` block of the output confirms.

There is one difference, and it is a genuine defect rather than a matter of taste.

The seeds `INT_MIN` and `INT_MAX` are not $-\infty$ and $+\infty$; they are the smallest and largest values an `int` can actually hold. So consider a tree consisting of the single node `INT_MIN`. The helper is entered with `lowerBound = INT_MIN`, and the first test is

$$\text{value} \leq \text{lowerBound} \;\Rightarrow\; \text{INT_MIN} \leq \text{INT_MIN} \;\Rightarrow\; \text{true},$$

so the function returns `false`. A perfectly good one-node BST is declared not to be a BST. The same happens for a tree containing `INT_MAX`. Because the two extreme values almost never appear in hand-made tests, this version passes every test a hurried programmer is likely to write.

Listing 8.9 in §8.14 contains exactly this narrow version, written for red-black trees, and demonstrates the failure; the relevant lines of its output are

```
The int-bounds pitfall, on a single BLACK node holding INT_MIN
  isBST with int bounds          : false
  isBSTRepaired, wider bounds    : true
```

There are three clean repairs.

1. **Widen the bound type.** Pass `long long` bounds and seed them with INT_MIN − 1 and INT_MAX + 1, values no `int` can equal. This is what **Listing 8.4** does. It is the smallest change, and the behavior on every other input is untouched.
2. **Let a bound be absent.** Pass `std::optional<int>` bounds, with an empty optional meaning "no bound on this side": `if (lowerBound && t->value <= *lowerBound) return false;`. Nothing is reserved as a sentinel, so nothing can collide with a real value.
3. **Switch to the inorder-walk method**, which needs no sentinel of its own as long as its "nothing seen yet" starting value is below every `int`. `isBSTByWalk` in **Listing 8.4** is this method.

> **HOW TO STATE THE REPAIR.** *"With `int` bounds, the sentinels `INT_MIN` and `INT_MAX` are attainable tree values, so a tree containing either is rejected. The bound parameters are therefore widened to `long long`; on every other input the behavior is unchanged."* Three sentences: the failing input, the cause, and the repair with its scope. That is what careful edge-case thinking looks like when it is written down.

8.6.8 The alternative method: the inorder walk

Chapter 6 §6.6.1 proved that a binary tree is a binary search tree **if and only if** its inorder walk is strictly increasing. That theorem gives a second, completely independent solution to Problem 8.3, and it is worth writing both: two methods agreeing is the cheapest correctness argument there is.

The method: walk the tree in order, remembering only the previously printed value, and fail the moment a value is not greater than its predecessor. `inorderIncreasing` in **Listing 8.4** is eight lines long. Two details matter.

The `prev` variable must be shared across the whole walk, not per call. In C++ that is `long long& prev`, a reference. A version that passes `prev` by value gets a function that compares each node only with nodes in its own subtree — and that is the local check again, in disguise.

The comparison must be `<=`, not `<`. With `<`, equal values pass, and the definition forbids them.

Cost comparison: both methods are $\Theta(n)$ time in the worst case and $O(h)$ stack space. The bounds method has one advantage worth stating — it can stop early at the *first* node that violates a bound, without having walked the whole left side of the tree first, which is why the diagnostic version can name the culprit. The walk method's advantage is that it needs no sentinel.

8.6.9 Common wrong turns on this problem

Ranked by how often they appear.

1. **The local check.** It gets the recursion and the base case right and the property wrong. §8.6.3.

2. **Non-strict bounds.** Writing `node.value < lowerBound` instead of `<=`. This accepts duplicates, which **Figure 8.1** forbids, and it is found immediately by the two-node test `20{20,.}`.
3. **Swapping the bounds in the recursive calls.** Writing `isBST(t.left, t.value, upperBound)` — the node's value as the *lower* bound on the *left*. This rejects every non-trivial tree, and it is caught in ten seconds by testing a three-node tree.
4. **Forgetting the base case**, so the code dereferences the empty tree. The program crashes on the first empty subtree it meets.
5. **Exposing the helper.** The bounds are an implementation detail; callers should see only `isBST(t)`. In **Listing 8.4** the helper is `static`, visible only inside its own file.
6. **Using `int` sentinels.** §8.6.7. The code is right on every input except those containing the two extreme values, which is exactly why it survives casual testing.

8.7 Problem 8.4 — `printDescending`: Largest First, With No List

8.7.1 The problem

Write a recursive function `void printDescending(MyTree t)` which receives a BST `t` of integers and prints its values in *descending order*. The function must work **without making a separate list of the values of `t`**.

8.7.2 What the problem is really about

One idea, and one restriction.

The idea is that **the inorder walk of a BST is sorted**, which Chapter 6 §6.6.1 proved, and which **Figure 8.2** illustrates: the inorder successor of 8 is 10, of 10 is 12, of 14 is 20. Read those relationships end to end and you have the sorted sequence. If the inorder walk — left, node, right — gives ascending order, then its mirror image — right, node, left — must give descending order. That is the whole answer, and it is four lines long.

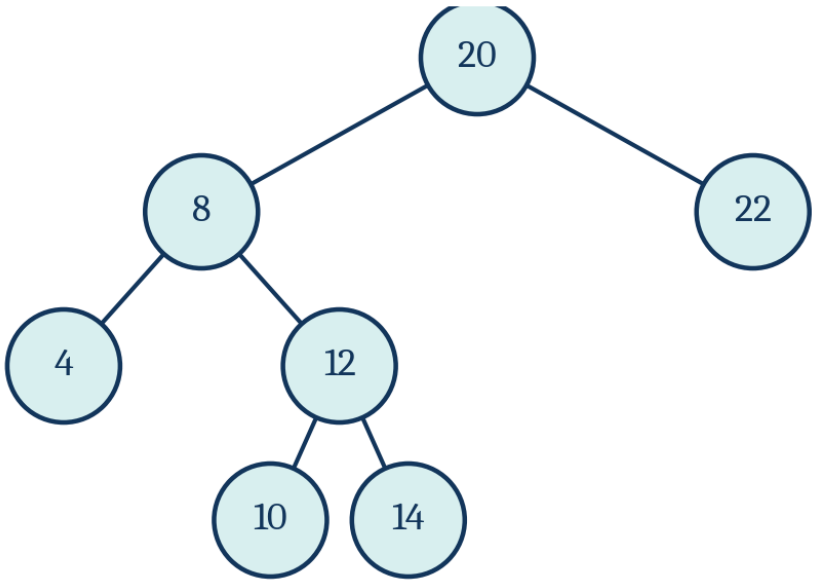


Figure 8.2 The inorder successor. Chaining the successors — 8, 10, 12, 14, 20 — traces the sorted order that the inorder walk produces. Reversing the chain gives descending order.

The restriction is the clause *"without making a separate list of values from `t`"*. That is not decoration. It rules out the solution most people reach for: collect the values into an array, sort or

reverse it, print it. The clause exists to force sorted output **from the structure of the tree alone**, which is the property the problem is about.

8.7.3 Deriving the answer

Step 1 — write down what the inorder walk does, and why it is sorted. For a tree $t = (v, L, R)$ that satisfies the BST property, every value in L is less than v and every value in R is greater than v. So if we print L's values in increasing order, then v, then R's values in increasing order, the whole output is increasing. By induction on the height, the inorder walk prints the values in increasing order.

Step 2 — reverse the requirement and see what changes. We want decreasing order. Every value in R is greater than v; every value in L is less than v. So if we print R's values in *decreasing* order, then v, then L's values in *decreasing* order, the whole output is decreasing.

Step 3 — write that as a recursion.

$$\begin{aligned}\text{printDescending}(\varnothing) &= \text{nothing}, \qquad \text{printDescending}((v, L, R)) \\ &= \text{printDescending}(R);\ \ \text{print } v;\ \ \text{printDescending}(L).\end{aligned}$$

Step 4 — check the restriction. Nothing is collected. Each value is printed at the instant the recursion reaches its node, and then forgotten. The only memory used is the recursion stack, which holds at most $h + 1$ frames — and the recursion stack is not "a separate list of values from `t`" by any reading; it is the mechanism by which *every* recursive traversal works, including the ascending one that nobody would call "a list".

Step 5 — the complexity. Every node is visited exactly once, and at each node the work is one comparison against `isEmpty`, one print and two calls: $\Theta(1)$. So the time is $\Theta(n)$. The space is $O(h)$ for the stack, which is $O(\lg n)$ if the tree is balanced and $O(n)$ if it is a chain.

> **A TRAVERSAL IS THREE STATEMENTS IN SOME ORDER.** Every one of the classic traversals is the same three statements — *recurse left*, *visit*, *recurse right* — in a different order. Preorder is visit, left, right. Inorder is left, visit, right. Postorder is left, right, visit. Reverse inorder is right, visit, left. There is nothing else to memorize. A request for "descending order" is a request to move one line.

8.7.4 The program

Listing 8.5 gives the recursive solution, the ascending walk for comparison, an iterative answer with an explicit stack, and a depth-traced version that demonstrates the $O(h)$ stack claim.

Listing 8.5 Problem 8.4 in full: the four-line recursive answer, the ascending walk beside it, the iterative answer with an explicit stack, and a trace confirming the $O(h)$ stack depth.

```
// printdesc.cpp -- Problem 8.4: printDescending().
#include <iostream>
#include <string>
#include <stack>
#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()); }

// ---- The answer: the mirror image of the inorder walk. ----
// Right subtree first, then the node, then the left subtree. No list is built:
// each value is printed the moment the recursion reaches it.
void printDescending(MyTree t) {
    if (isEmpty(t)) return;        // base case: nothing to print
    printDescending(t->right);     // every value here is GREATER than t->value
    cout << t->value << ' ';       // then t itself
    printDescending(t->left);      // every value here is SMALLER than t->value
}

// For comparison, the ordinary ascending walk.
void printAscending(MyTree t) {
    if (isEmpty(t)) return;
    printAscending(t->left);
    cout << t->value << ' ';
    printAscending(t->right);
}
```

Listing 8.5 (Continued) *Problem 8.4 in full: the four-line recursive answer, the ascending walk beside it, the iterative answer with an explicit stack, and a trace confirming the O(h) stack depth.*

```
// ---- Alternative answer: the same traversal with an explicit stack. ----
// Uses O(h) extra space, which is what the recursion uses anyway; it still
// builds no list of all n values.
void printDescendingIterative(MyTree t) {
    stack<MyTree> s;
    MyTree cur = t;
    while (cur != nullptr || !s.empty()) {
        while (cur != nullptr) { s.push(cur); cur = cur->right; }  // dive right
        cur = s.top(); s.pop();
        cout << cur->value << ' ';
        cur = cur->left;                                           // then left
    }
}

// A depth counter, to show that the recursion never nests deeper than h+1.
int maxDepth = 0;
void printDescendingTraced(MyTree t, int d) {
    if (isEmpty(t)) return;
    maxDepth = max(maxDepth, d);
    printDescendingTraced(t->right, d + 1);
    printDescendingTraced(t->left,  d + 1);
}

int main() {
    // The same eight-node tree used throughout the chapter.
    MyTree t = node(13,
                    node(10, node(5, leaf(4), leaf(6)), leaf(11)),
                    node(15, emptyTree(), leaf(16)));

    cout << "ascending  : "; printAscending(t);            cout << "\n";
    cout << "descending : "; printDescending(t);           cout << "\n";
    cout << "iterative  : "; printDescendingIterative(t);  cout << "\n";

    printDescendingTraced(t, 1);
    cout << "tree height h = " << height(t)
         << ", deepest recursive call = " << maxDepth << "\n";

    cout << "single node: "; printDescending(leaf(42));    cout << "\n";
    cout << "empty tree : "; printDescending(emptyTree()); cout << "(nothing)\n";
```

Listing 8.5 (Continued) *Problem 8.4 in full: the four-line recursive answer, the ascending walk beside it, the iterative answer with an explicit stack, and a trace confirming the O(h) stack depth.*

```
    // A right-leaning chain 1 -> 2 -> ... -> 6: descending must print 6..1.
    MyTree chain = emptyTree();
    for (int v = 6; v >= 1; --v) chain = node(v, emptyTree(), chain);
    cout << "chain 1..6 : "; printDescending(chain);       cout << "\n";
    return 0;
}
```

The program prints:

```
ascending  : 4 5 6 10 11 13 15 16
descending : 16 15 13 11 10 6 5 4
iterative  : 16 15 13 11 10 6 5 4
tree height h = 3, deepest recursive call = 4
single node: 42
empty tree : (nothing)
chain 1..6 : 6 5 4 3 2 1
```

Three things to read off that output. The descending line is the ascending line backward, as it must be. The iterative line is identical to the recursive one, which is the cheapest possible check

that the alternative method is right. And the deepest recursive call is 4 on a tree of height 3, confirming the bound $depth \leq h + 1$: there are $h + 1$ levels of nodes, and the calls on empty subtrees below the leaves do no work and are counted as returning immediately.

8.7.5 The alternative method: an explicit stack

`printDescendingIterative` in **Listing 8.5** deserves a walk-through, because the iterative inorder walk is a standard interview question and the reverse version is its mirror.

The invariant is: *the stack holds exactly those nodes whose value has not yet been printed but all of whose right-subtree values have been printed.* Reading the loop:

1. The inner `while` pushes `cur`, then moves to `cur->right`, repeatedly. It therefore pushes the whole right spine from `cur` downward, ending with `cur == nullptr`.
2. `s.top()` is then the deepest node on that spine — the largest value not yet printed. Pop it and print it.
3. Set `cur` to that node's left child and go round again. The left child's own right spine is pushed next, and so on.

Does this violate the "no separate list" clause? No: the stack holds *nodes still to be processed*, never more than $h + 1$ of them, and it is exactly the data the recursion keeps implicitly. It is still worth saying so in a comment, because a reader skimming for an array will see `stack<MyTree>` and pause: *"the stack holds at most $h + 1$ pending nodes, never a list of the n values; it is the recursion stack made explicit."*

8.7.6 Common wrong turns on this problem

1. **Building a list anyway.** Collecting into a `vector`, calling `sort`, then `reverse`. This is the forbidden method, and it also throws away the only interesting property of the input — that it is already sorted by structure. The output is right and the solution is not.
2. **Printing ascending.** Writing the ordinary inorder walk, which is caught by looking at the first line of output.
3. **Getting the order right but the parentheses wrong** — writing `printDescending(t->right); printDescending(t->left); cout << t->value;`, which is a postorder walk and prints nonsense. Test on the three-node tree 10{5,15}: the answer must be `15 10 5`.
4. **Returning a `string` instead of printing.** The signature says `void`. Building and returning a string is, in effect, building a list.
5. **Forgetting the base case** and dereferencing the empty tree.

8.8 Problem 8.5 — `max`: the Largest Key Along One Path

8.8.1 The problem

Write an **efficient** recursive function `int max(MyTree t)` which receives a BST `t` of integers and returns its largest value. **It must not visit and compare all the nodes of the tree**: it may follow at most one path from the root, so that on a balanced BST it runs in $O(\lg n)$ time.

Hint: stand at a node `x` of a BST. Every value in the left subtree of `x` is less than or equal to ($\leq$) the value of `x`. So the maximum cannot be in the left subtree. Where is it?

8.8.2 What the problem is really about

Not "can you find a maximum" — a first-week programmer can do that. It is about whether you can **use the BST property to avoid work**. The words *"must not visit and compare all the nodes"* turn a trivial task into a real one: a correct result obtained by visiting all n nodes is explicitly ruled out, so the substance of the problem is the pruning argument, not the number.

8.8.3 Deriving the answer

Step 1 — state the fact we have. At any node x with value v, every value in the left subtree of x is less than v, and every value in the right subtree is greater than v. (The hint says $\leq$; the definition of **Figure 8.1** says $<$ and forbids duplicates. Either way the argument below is unaffected, and §8.8.6 explains why.)

Step 2 — ask where the maximum can be. The tree's values are: v itself, the values in the left subtree, and the values in the right subtree. Every left-subtree value is $<v$, so no left-subtree value can be the maximum as long as v is present — and v always is. **The left subtree can be discarded, unexamined, at every node.**

Step 3 — compare v with the right subtree. Every right-subtree value is $>v$. So if the right subtree is non-empty, the maximum is *not* v; it is the maximum of the right subtree. And if the right subtree is empty, then the only candidates left are v and the left-subtree values, all of which are smaller, so the answer is v.

Step 4 — write it.

$$\max((v, L, \varnothing)) = v, \qquad \max((v, L, R)) = \max(R) \text{ when } R \neq \varnothing.$$

The base case is *"the right subtree is empty"*, not *"the tree is empty"*. The empty tree has no maximum at all — there is no integer to return — and §8.8.7 says what to do about that.

Step 5 — check that "one path" is honored. Each call makes at most one recursive call, and it is always on the right subtree. So the sequence of nodes visited is t, t's right child, that node's right child, and so on, until the right child is empty. That is the **rightmost path** of the tree — one path, from the root, exactly as required.

> **DEFINITION (the right spine).** The **right spine** of a non-empty tree is the sequence of nodes obtained by starting at the root and repeatedly taking the right child until the right child is empty. Its length is at most $h + 1$ nodes. `max` returns the value at the end of the right spine; `min` returns the value at the end of the left spine.

Step 6 — count the cost. The right spine has at most $h + 1$ nodes, one per level. Each node costs $\Theta(1)$: one emptiness test and one recursive call. So the running time is $O(h)$ where h is the height. That is the honest bound, and §8.8.5 explains why it is not quite what the problem statement says.

8.8.4 The program

Listing 8.6 implements the solution recursively, again iteratively, and — for contrast — the forbidden full-scan version, with a counter so that the difference is visible rather than asserted.

Listing 8.6 Problem 8.5 in full, with a node-visit counter that makes the difference between the permitted and the forbidden method measurable.

```
// bstmax.cpp -- Problem 8.5: max() on a BST in O(h) time.
#include <iostream>
#include <string>
#include <stdexcept>
#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()); }

long long visits = 0;      // counts how many nodes a routine actually looks at

// ---- The answer: walk right until the right subtree is empty. ----
int maxBST(MyTree t) {
    if (isEmpty(t)) throw invalid_argument("max() of an empty tree is undefined");
    ++visits;
    if (isEmpty(t->right)) return t->value;   // no larger value can exist
    return maxBST(t->right);                  // ONE recursive call, never two
}

// The mirror image, for free.
int minBST(MyTree t) {
    if (isEmpty(t)) throw invalid_argument("min() of an empty tree is undefined");
    ++visits;
    if (isEmpty(t->left)) return t->value;
    return minBST(t->left);
}

// ---- Alternative answer: the same walk without recursion. ----
int maxBSTIterative(MyTree t) {
    if (isEmpty(t)) throw invalid_argument("max() of an empty tree is undefined");
    while (!isEmpty(t->right)) { ++visits; t = t->right; }
    ++visits;
    return t->value;
}
```

Listing 8.6 (Continued) *Problem 8.5 in full, with a node-visit counter that makes the difference between the permitted and the forbidden method measurable.*

```
// ---- The forbidden answer: look at every node. Correct, but disallowed. ----
int maxByFullScan(MyTree t) {
    if (isEmpty(t)) throw invalid_argument("max() of an empty tree is undefined");
    ++visits;
    int best = t->value;
    if (!isEmpty(t->left))  best = max(best, maxByFullScan(t->left));
    if (!isEmpty(t->right)) best = max(best, maxByFullScan(t->right));
    return best;
}

// Print the rightmost path, so the reader can see which nodes were touched.
string rightSpine(MyTree t) {
    string s;
    while (!isEmpty(t)) { s += to_string(t->value); t = t->right; if (!isEmpty(t)) s += " -
> "; }
    return s;
}

int countNodes(MyTree t) { return isEmpty(t) ? 0 : 1 + countNodes(t->left) + countNodes(t-
>right); }

// Build the perfectly balanced BST on 1..n.
MyTree perfect(int lo, int hi) {
    if (lo > hi) return emptyTree();
    int mid = lo + (hi - lo) / 2;
    return node(mid, perfect(lo, mid - 1), perfect(mid + 1, hi));
}

int main() {
    MyTree t = perfect(1, 15);            // 15 nodes, height 3
    cout << "tree        : perfectly balanced BST on 1..15\n";
    cout << "nodes n     : " << countNodes(t) << ",  height h = " << height(t) << "\n";
    cout << "right spine : " << rightSpine(t) << "\n\n";

    visits = 0; int a = maxBST(t);
    cout << "maxBST          -> " << a << "   nodes visited: " << visits << "\n";
    visits = 0; int b = maxBSTIterative(t);
    cout << "maxBSTIterative -> " << b << "   nodes visited: " << visits << "\n";
    visits = 0; int c = maxByFullScan(t);
    cout << "maxByFullScan   -> " << c << "   nodes visited: " << visits << "
(forbidden)\n";
    visits = 0; int d = minBST(t);
    cout << "minBST          -> " << d << "   nodes visited: " << visits << "\n\n";
```

Listing 8.6 (Continued) *Problem 8.5 in full, with a node-visit counter that makes the difference between the permitted and the forbidden method measurable.*

```
    // The honest worst case: a left-leaning chain. h = n - 1, so O(h) is O(n).
    MyTree chain = emptyTree();
    for (int v = 1; v <= 10; ++v) chain = node(v, chain, emptyTree());
    cout << "left-leaning chain 10,9,...,1 (h = " << height(chain) << ")\n";
    visits = 0; cout << "  maxBST -> " << maxBST(chain) << "   nodes visited: " << visits
<< "\n";
    visits = 0; cout << "  minBST -> " << minBST(chain) << "   nodes visited: " << visits
<< "\n\n";

    try { maxBST(emptyTree()); }
    catch (const exception& e) { cout << "empty tree  : " << e.what() << "\n"; }
    return 0;
}
```

Its output:

```
tree        : perfectly balanced BST on 1..15
nodes n     : 15,  height h = 3
right spine : 8 -> 12 -> 14 -> 15

maxBST          -> 15   nodes visited: 4
maxBSTIterative -> 15   nodes visited: 4
maxByFullScan   -> 15   nodes visited: 15  (forbidden)
minBST          -> 1   nodes visited: 4

left-leaning chain 10,9,...,1 (h = 9)
  maxBST -> 10   nodes visited: 1
  minBST -> 1   nodes visited: 10

empty tree  : max() of an empty tree is undefined
```

Four nodes against fifteen, on a fifteen-node tree of height 3: the permitted method visits $h + 1 = 4$ nodes and the forbidden one visits all $n = 15$. That ratio is the whole point of the problem, and a measured count like this is far more persuasive than the sentence "this version is more efficient".

The chain block is the honest part. On a left-leaning chain of ten nodes, `maxBST` visits **one** node — the root is already the largest — while `minBST` visits all ten, because the left spine *is* the whole tree. Same algorithm, same tree, ten times the work. That is what $O(h)$ means, and it is why the next subsection matters.

8.8.5 $O(h)$, $O(\lg n)$, and how to state the bound so that it is true

The problem says the function should run *"in $O(\lg n)$ time on a balanced BST"*.

Read the qualifier. *Balanced* — the statement quietly depends on the tree being short, which the AVL trees of the neighboring problems guarantee. For an arbitrary binary search tree the claim is false, and the chain in **Listing 8.6** is the counterexample: $n = 10$, $h = 9$, and `minBST` visits 10 nodes, which is $\Theta(n)$, not $\Theta(\lg n)$.

The correct chain of statements, in the language of Chapter 3, is:

1. `max` runs in $O(h)$ time, where h is the height of the tree. **This is always true.**
2. For a binary search tree with no balance guarantee, h can be as large as $n - 1$, so the bound degrades to $O(n)$. Chapter 6 §6.13 measured exactly this.
3. For an **AVL** tree, Chapter 6 §6.15.2 proved $h \leq 1.4405\lg(n + 2) - 1.3277 = O(\lg n)$. Substituting into statement 1 gives $O(\lg n)$.
4. Therefore: `max` runs in $O(\lg n)$ time **on a height-balanced tree**, and in $O(h)$ time in general.

> **HOW TO STATE THE BOUND SO THAT IT IS TRUE.** *"`max()` visits only the right spine, so its cost is $O(h)$ where h is the tree's height. For the AVL trees of Problems 8.6 to 8.10, $h \leq 1.4405\lg(n + 2) - 1.3277$, so the cost is $O(\lg n)$. For an unbalanced BST the same code is $O(n)$ — the test on a ten-node left-leaning chain visits all ten nodes — so the $O(\lg n)$ claim depends on the balance guarantee, not on the code."*
>
> That paragraph is completely correct, and it shows the difference between a property of an algorithm and a property of its input — a distinction that is worth more than any single bound.

8.8.6 The $\leq$ in the hint

The hint says left-subtree values are "less than or equal ($\leq$)" to the node's value, while **Figure 8.1** says "$<$" and that no two nodes share a value. These are two different BST conventions:

- **Strict BST**, no duplicates: the definition used in this book, and what `isBST` in **Listing 8.4** enforces.

- **Non-strict BST**, duplicates allowed on the left: a common alternative, used by some textbooks.

Does it change the solution of Problem 8.5? No. Under either convention, no left-subtree value *exceeds* the node's value, and that is the only fact the derivation of §8.8.3 uses. The maximum is at the end of the right spine either way. Under the non-strict convention the maximum value may occur more than once, but `max` returns the value, not a node, so the returned integer is the same.

In one line: *the hint's $\leq$ and the definition's $<$ give the same answer here, because the derivation only needs "no left-subtree value exceeds the root".*

8.8.7 The empty tree

`int max(MyTree t)` returns an `int`. There is no `int` that means "there is no maximum". So what should `max(empty)` do? Three defensible answers, and you should name the one you chose:

1. **Throw.** `throw invalid_argument("max() of an empty tree is undefined")`. This is what **Listing 8.6** does, and it is the honest choice: the function's contract is undefined on the empty tree, so it should say so loudly.
2. **Return `INT_MIN`** as a sentinel. Cheap, and wrong for the same reason §8.6.7 was wrong: `INT_MIN` is a legal tree value, so the caller cannot distinguish "empty" from "the tree contains the smallest int".
3. **Document a precondition** — "the caller must ensure `t` is non-empty" — and leave the behavior undefined. Acceptable if written down; a silent crash is not.

Any of the three is defensible with a sentence attached. What is not defensible is code that dereferences the empty tree with no comment, because that is indistinguishable from not having thought about it.

8.8.8 Common wrong turns on this problem

1. **Recursing into both subtrees.** `return max(t->value, max(left), max(right))`. Correct output, explicitly ruled-out method, and it visits all n nodes. §8.8.4 measures it.
2. **Using the base case "the tree is empty" instead of "the right subtree is empty".** This gives `if (isEmpty(t)) return INT_MIN; return std::max(t->value, max(t->right));` — which works, but calls itself once more than necessary and reintroduces the sentinel problem. Prefer the clean base case.
3. **Walking left.** A transposition typo. Caught in five seconds by testing the three-node tree 10{5,15}, whose max is 15.
4. **Claiming $O(\lg n)$ with no justification.** §8.8.5 shows how to state the bound so that it is true.
5. **Assuming the tree is an AVL tree when the problem says only "BST"**, and then being unable to answer the follow-up *"what if it is a chain?"*

8.9 Problem 8.6 — `isHeightBalanced`: the AVL Condition in One Pass

8.9.1 The problem

Write a recursive function `bool isHeightBalanced(MyTree t)` which checks whether the tree `t` is height-balanced — whether it satisfies the AVL condition. Aim for a single pass over the tree.

8.9.2 What the AVL condition is

Chapter 6 states it: *for each node, the height difference between the left and right branches is at most 1*. **Figure 8.3** shows a tree that satisfies it and **Figure 8.4** one that does not.

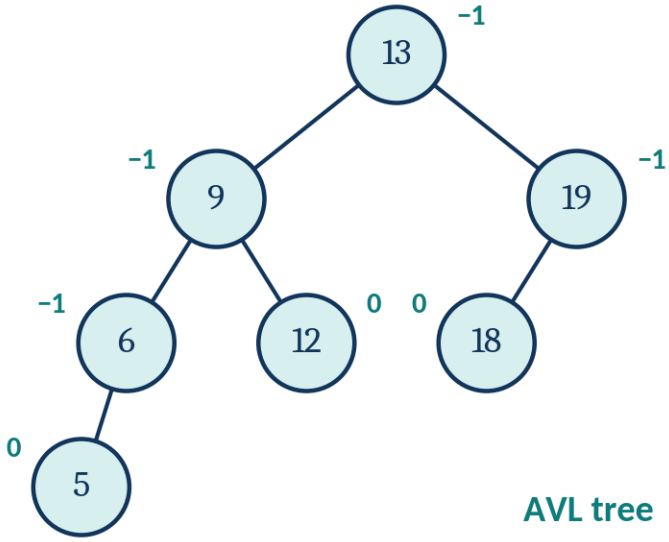


Figure 8.3 A tree that **is** an AVL tree. Every node's two subtrees differ in height by at most one.

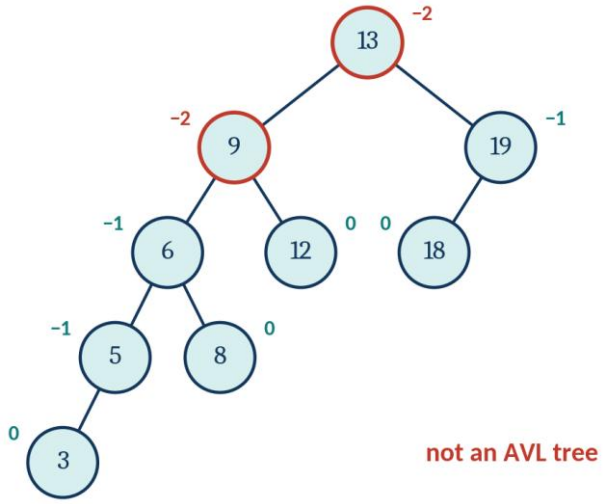


Figure 8.4 A tree that is **not** an AVL tree. Hanging the node 3 below 5 pushes two balance factors, those of 9 and 13, to −2, out of the legal range.

In our notation, from §8.5:

> **DEFINITION (height-balanced).** A tree t is **height-balanced** — an AVL tree — when

$$|\mathrm{bf}(x)| \leq 1 \quad \text{for every node } x \text{ of } t, \tag{8.9}$$

> where $\mathrm{bf}(x) = h(\mathrm{right}(x)) - h(\mathrm{left}(x))$ and the empty tree has height -1. Equivalently: $\mathrm{bf}(x) \in \{-1, 0, +1\}$ at every node. The empty tree is height-balanced, vacuously.

Work **Figure 8.4** by hand, using the tree Chapter 6 §6.14 described: root 13; left child 9 whose children are 6 and 12; node 6 with left child 5 and right child 8; the new node 3 hung below 5; right child 19 with a left child 18.

- Node 3: a leaf. $h = 0$, $\mathrm{bf} = -1 - (-1) = 0$. ✓

- Node 5: left subtree is the leaf 3 ($h = 0$), right subtree empty ($h = -1$). $\text{bf} = -1 - 0 = -1$. ✓ And $h(5) = 1 + \max(0, -1) = 1$.
- Node 8: a leaf. $h = 0$, $\text{bf} = 0$. ✓
- Node 6: left subtree rooted at 5 ($h = 1$), right subtree the leaf 8 ($h = 0$). $\text{bf} = 0 - 1 = -1$. ✓ And $h(6) = 1 + \max(1,0) = 2$.
- Node 12: a leaf. $h = 0$, $\text{bf} = 0$. ✓
- Node 9: left subtree rooted at 6 ($h = 2$), right subtree the leaf 12 ($h = 0$). $\text{bf} = 0 - 2 = \mathbf{-2}$. ✗ **Out of range.**
- Node 13: left subtree rooted at 9 ($h = 3$), right subtree rooted at 19 ($h = 1$). $\text{bf} = 1 - 3 = \mathbf{-2}$. ✗ **Out of range too.**

So the tree is not height-balanced, and the first offending node encountered from the bottom is 9 — not 6, as a quick glance at the long left branch might suggest: node 6 is still balanced because its right child 8 props it up. Hold on to that number; the program will find it.

8.9.3 The obvious answer, and what it costs

The definition Equation (8.9) says "for every node, the difference of the subtree heights". So the obvious code writes exactly that:

```
int heightSlow(MyTree t) {
    if (isEmpty(t)) return -1;
    return 1 + max(heightSlow(t->left), heightSlow(t->right));
}
bool isHeightBalancedSlow(MyTree t) {
    if (isEmpty(t)) return true;
    int hl = heightSlow(t->left), hr = heightSlow(t->right);
    if (abs(hl - hr) > 1) return false;
    return isHeightBalancedSlow(t->left) && isHeightBalancedSlow(t->right);
}
```

This is **correct**. It passes every test of correctness. What it fails is the request for a single pass, and that failure is the interesting part, because the code recomputes the same heights over and over. At the root it computes the heights of both subtrees, which together inspect $n - 1$ nodes. Then it recurses into the left subtree and computes the heights of *its* two subtrees — inspecting nodes it has already inspected. And so on, all the way down.

Let us count exactly. At each node x that the recursion reaches, the two calls to `heightSlow` inspect every node strictly below x — that is $\text{size}(x) - 1$ nodes — plus one inspection for x itself in the balance check. So if the tree passes and every node is reached, the total number of node inspections is

$$W(T) = \sum_{x \in T} [(\text{size}(x) - 1) + 1] = \sum_{x \in T} \text{size}(x).$$

That sum has a closed form for a **perfect** tree. Let the tree be perfect of height k, so $n = 2^{k+1} - 1$ nodes, with 2^{ℓ} nodes at level ℓ for $\ell = 0,1, \dots, k$, and each node at level ℓ rooting a subtree of size $2^{k-\ell+1} - 1$. Then

$$\sum_{x \in T} \text{size}\,(x) = \sum_{\ell=0}^{k} 2^{\ell}\,(2^{k-\ell+1} - 1) = \sum_{\ell=0}^{k} (2^{k+1} - 2^{\ell})$$
$$= (k+1)2^{k+1} - (2^{k+1} - 1).$$

Now substitute $2^{k+1} = n + 1$ and $k + 1 = \lg(n + 1)$:

$$W = (n + 1)\lg(n + 1) - n. \tag{8.10}$$

Check it on the smallest cases. For $k = 2$, $n = 7$: $W = 8 \times 3 - 7 = 17$. For $k = 3$, $n = 15$: $W = 16 \times 4 - 15 = 49$. For $n = 127$: $W = 128 \times 7 - 127 = 769$. The program in **Listing 8.7** prints 17, 49 and 769 for exactly those trees, and **Listing 8.14** in §8.23 checks the formula against the program for every perfect tree up to $n = 2047$.

So the obvious method costs $\Theta(n\lg n)$ on a balanced tree. The one-pass method below costs $\Theta(n)$. The saving is a factor of $\lg n$ — for $n = 10^6$, a factor of twenty.

> **A CLAIM YOU WILL SEE EVERYWHERE, AND IT IS NOT TRUE HERE.** It is very commonly asserted that the naive balance check is $\Theta(n^2)$. It is not — not for this predicate. The quadratic case would need $\Theta(n)$ nodes each costing $\Theta(n)$, which needs a tall tree; but a tree that *passes* the balance check has $h = O(\lg n)$, and a tree that *fails* it makes the check return early. The measured worst case for `isHeightBalancedSlow` is $\Theta(n\lg n)$, and the derivation above gives the exact constant. Where $\Theta(n^2)$ genuinely does appear is in the closely related routine that labels *every* node with its height by calling `heightSlow` once per node, with no early exit: that costs $\sum_x \text{size}\,(x)$ over a possibly tall tree, which on a chain of n nodes is $n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2} = \Theta(n^2)$. **Listing 8.7** measures that too, and gets 8256 for $n = 128$ — precisely $\frac{128 \times 129}{2}$.

8.9.4 The answer: one pass, with a sentinel

The waste has an obvious cause: `isHeightBalancedSlow` throws away every height it computes. Chapter 7's lesson was *never recompute what you already know*. Apply it.

Step 1 — notice that the recursion already visits every node once. If each visit could *return* the subtree's height, the parent would get both heights for free.

Step 2 — notice that the parent also needs a yes/no answer. So one recursive call must deliver two things: the height, and whether the subtree is balanced.

Step 3 — pack both into one integer. Heights are -1 or greater. So any value below -1 is not a possible height and can be used to mean "not balanced". Choose -2.

Step 4 — write the helper.

$$\text{ch}(\emptyset) = -1; \qquad \text{ch}((v, L, R)) = \begin{cases} -2 & \text{if ch}(L) = -2 \\ -2 & \text{if ch}(R) = -2 \\ -2 & \text{if } |\text{ch}(R) - \text{ch} \\ 1 + \max(\text{ch}(L), \text{ch}(R)) & \text{otherwise.} \end{cases} \tag{8.11}$$

Step 5 — the answer is isHeightBalanced$(t) \Leftrightarrow \text{ch}(t) \neq -2$, with ch as in Equation (8.11).

Step 6 — correctness. Two directions. If the tree is balanced, then no node ever triggers a -2, so every call returns $1 +$ max of its children's true heights, which by induction on the height is the

node's true height; so $\mathrm{ch}(t) = h(t) \neq -2$. If some node is unbalanced, take the *lowest* such node x: every node below x is balanced, so the calls below x return true heights, so the test at x sees the true heights, sees a difference exceeding 1, and returns -2; and -2 then propagates upward through the first two cases to the root.

Step 7 — the cost. Each node is entered exactly once and does $\Theta(1)$ work, so the time is $\Theta(n)$ in the worst case, and less when the early exit fires. The space is $O(h)$ for the stack. This is optimal: you cannot decide whether *every* node is balanced without, in the worst case, looking at every node.

> **THE SENTINEL PATTERN, WORTH REMEMBERING FOR LIFE.** When a recursion must return a value *and* a failure flag, and the value's type has an unused element, use that element as the flag instead of returning a pair. Heights are ≥ -1, so -2 is free. Sizes are ≥ 0, so -1 is free. This is the same device as returning -1 from a search that finds nothing. It appears again in `computeBlackHeight` of Problem 8.13 in §8.14, where -1 is the flag because black-heights are ≥ 0 — the same technique on a different invariant, which is the surest sign that it is a technique and not a trick.

8.9.5 The program

Listing 8.7 contains the slow method, the one-pass method, the height-annotation routine that genuinely is quadratic, a diagnostic that names the offending node, and a measured comparison across tree shapes.

Listing 8.7 Problem 8.6 in full: the wasteful method, the one-pass sentinel method, the genuinely quadratic height-annotation routine, and a measured comparison over eleven tree shapes.

```
// isbalanced.cpp -- Problem 8.6: isHeightBalanced().
#include <iostream>
#include <iomanip>
#include <string>
#include <algorithm>
using namespace std;

struct MyTreeNode;
using MyTree = MyTreeNode*;
struct MyTreeNode { int value; MyTree left, right; };   // NO cached height here:
                                                        // the question is whether
                                                        // we can compute it cheaply
MyTree emptyTree()       { return nullptr; }
bool   isEmpty(MyTree t) { return t == nullptr; }
MyTree node(int v, MyTree l, MyTree r) {
    MyTree t = new MyTreeNode; t->value = v; t->left = l; t->right = r; return t;
}
MyTree leaf(int v) { return node(v, emptyTree(), emptyTree()); }

long long work = 0;        // every node inspection, anywhere, is counted here

// ---- The obvious but wasteful route: recompute the height at every node. ----
int heightSlow(MyTree t) {
    if (isEmpty(t)) return -1;
    ++work;
    return 1 + max(heightSlow(t->left), heightSlow(t->right));
}
bool isHeightBalancedSlow(MyTree t) {
    if (isEmpty(t)) return true;
    ++work;
    int hl = heightSlow(t->left), hr = heightSlow(t->right);
    if (abs(hl - hr) > 1) return false;
    return isHeightBalancedSlow(t->left) && isHeightBalancedSlow(t->right);
}

// ---- The answer: one pass. Return the height, or -2 to mean "not balanced". ----
// -1 is already taken (the height of the empty tree), so the failure flag must
// be a value no real height can ever have.
const int UNBALANCED = -2;
static int checkedHeight(MyTree t) {
    if (isEmpty(t)) return -1;                  // base case
    ++work;
    int hl = checkedHeight(t->left);
    if (hl == UNBALANCED) return UNBALANCED;    // give up early
    int hr = checkedHeight(t->right);
    if (hr == UNBALANCED) return UNBALANCED;
```

Listing 8.7 (Continued) *Problem 8.6 in full: the wasteful method, the one-pass sentinel method, the genuinely quadratic height-annotation routine, and a measured comparison over eleven tree shapes.*

```
    if (abs(hl - hr) > 1)  return UNBALANCED;    // this node breaks the invariant
    return 1 + max(hl, hr);                      // otherwise, the real height
}
bool isHeightBalanced(MyTree t) { return checkedHeight(t) != UNBALANCED; }

// ---- Diagnostic form: which node breaks it, and by how much. ----
static int auditHeight(MyTree t, string& why) {
    if (isEmpty(t)) return -1;
    int hl = auditHeight(t->left,  why); if (hl == UNBALANCED) return UNBALANCED;
    int hr = auditHeight(t->right, why); if (hr == UNBALANCED) return UNBALANCED;
    if (abs(hl - hr) > 1) {
        why = "node " + to_string(t->value) + ": left height " + to_string(hl)
            + ", right height " + to_string(hr) + ", bf = " + to_string(hr - hl);
        return UNBALANCED;
    }
    return 1 + max(hl, hr);
}

// ---- The same waste, isolated: label every node with its own height by
// ---- calling heightSlow() once per node. Cost = sum over v of size(v).
void annotateHeights(MyTree t) {
    if (isEmpty(t)) return;
    heightSlow(t);                       // recomputes the whole subtree, every time
    annotateHeights(t->left);
    annotateHeights(t->right);
}

int countNodes(MyTree t) { return isEmpty(t) ? 0 : 1 + countNodes(t->left) + countNodes(t-
>right); }

void report(const char* name, MyTree t) {
    string why = "-";
    bool ok = auditHeight(t, why) != UNBALANCED;
    work = 0; bool s = isHeightBalancedSlow(t);  long long wSlow = work;
    work = 0; bool f = isHeightBalanced(t);      long long wFast = work;
    work = 0; annotateHeights(t);                long long wAnn  = work;
    cout << left << setw(26) << name
         << " n=" << setw(4) << countNodes(t)
         << " slow:" << (s ? "yes" : "no ") << " w=" << setw(6) << wSlow
         << " fast:" << (f ? "yes" : "no ") << " w=" << setw(4) << wFast
         << " annotate w=" << setw(6) << wAnn << "\n";
    if (!ok) cout << string(26, ' ') << " first fault: " << why << "\n";
}
```

Listing 8.7 (Continued) *Problem 8.6 in full: the wasteful method, the one-pass sentinel method, the genuinely quadratic height-annotation routine, and a measured comparison over eleven tree shapes.*

```
// Perfectly balanced BST on lo..hi.
MyTree perfect(int lo, int hi) {
    if (lo > hi) return emptyTree();
    int mid = lo + (hi - lo) / 2;
    return node(mid, perfect(lo, mid - 1), perfect(mid + 1, hi));
}
// Right-leaning chain 1 -> 2 -> ... -> n.
MyTree chain(int n) {
    MyTree t = emptyTree();
    for (int v = n; v >= 1; --v) t = node(v, emptyTree(), t);
    return t;
}

int main() {
    // The AVL tree of the figure: root 13, left 9{6{5,.},12}, right 19{18,.}
    MyTree avl = node(13, node(9, node(6, leaf(5), emptyTree()), leaf(12)),
                          node(19, leaf(18), emptyTree()));
    // Node 8 under 6, and 3 hung under 5 -- the "not AVL" figure.
    MyTree notAvl = node(13, node(9, node(6, node(5, leaf(3), emptyTree()), leaf(8)),
leaf(12)),
                             node(19, leaf(18), emptyTree()));

    cout << "legend: w = node inspections;  slow = recompute heights;  "
            "fast = one pass with a sentinel\n";
    report("the AVL example",          avl);
    report("the non-AVL example",      notAvl);
    report("empty tree",               emptyTree());
    report("single node",              leaf(7));
    for (int n : {7, 15, 31, 63, 127}) {
        string nm = "perfect tree, n = " + to_string(n);
        report(nm.c_str(), perfect(1, n));
    }
    for (int n : {8, 16, 32, 64, 128}) {
        string nm = "chain, n = " + to_string(n);
        report(nm.c_str(), chain(n));
    }
    return 0;
}
```

Its output:

```
legend: w = node inspections;  slow = recompute heights;  fast = one pass with a sentinel
the AVL example            n=7    slow:yes w=18     fast:yes w=7    annotate w=18
the non-AVL example        n=9    slow:no  w=9      fast:no  w=7    annotate w=27
                           first fault: node 9: left height 2, right height 0, bf = -2
empty tree                 n=0    slow:yes w=0      fast:yes w=0    annotate w=0
single node                n=1    slow:yes w=1      fast:yes w=1    annotate w=1
perfect tree, n = 7        n=7    slow:yes w=17     fast:yes w=7    annotate w=17
perfect tree, n = 15       n=15   slow:yes w=49     fast:yes w=15   annotate w=49
perfect tree, n = 31       n=31   slow:yes w=129    fast:yes w=31   annotate w=129
perfect tree, n = 63       n=63   slow:yes w=321    fast:yes w=63   annotate w=321
perfect tree, n = 127      n=127  slow:yes w=769    fast:yes w=127  annotate w=769
chain, n = 8               n=8    slow:no  w=8      fast:no  w=8    annotate w=36
                           first fault: node 6: left height -1, right height 1, bf = 2
chain, n = 16              n=16   slow:no  w=16     fast:no  w=16   annotate w=136
                           first fault: node 14: left height -1, right height 1, bf = 2
chain, n = 32              n=32   slow:no  w=32     fast:no  w=32   annotate w=528
                           first fault: node 30: left height -1, right height 1, bf = 2
chain, n = 64              n=64   slow:no  w=64     fast:no  w=64   annotate w=2080
                           first fault: node 62: left height -1, right height 1, bf = 2
chain, n = 128             n=128  slow:no  w=128    fast:no  w=128  annotate w=8256
                           first fault: node 126: left height -1, right height 1, bf = 2
```

Line 3 confirms the hand calculation of §8.9.2: the first fault in the tree of **Figure 8.4** is at **node 9**, with left height 2, right height 0 and balance factor −2. That is precisely what we computed by hand.

Table 8.6 collects the measured numbers against the closed forms derived above.

Table 8.6 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

Every entry in the two "measured" columns equals its closed form. That is what it means to verify an analysis rather than assert it, and a table like this is the strongest possible evidence that a complexity claim is right.

8.9.6 The recurrence, solved with the Master Theorem

For completeness, and because Chapter 3 built the machinery for exactly this, here is the same result from the recurrence.

For the **one-pass** method on a tree in which each subtree holds about half the nodes:

$$T(n) = 2T(n/2) + \Theta(1). \tag{8.12}$$

Match Equation (8.12) against the Master Theorem's form $T(n) = aT(n/b) + f(n)$: $a = 2$, $b = 2$, $f(n) = \Theta(1)$. Then $\log_b a = \log_2 2 = 1$, so $n^{\log_b a} = n$. Compare $f(n) = \Theta(1)$ with n^1: we need $f(n) = O(n^{1-\varepsilon})$ for some $\varepsilon > 0$, and $\varepsilon = 1$ works, since $\Theta(1) = O(n^0)$. That is **case 1**, so

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n).$$

For the **slow** method, the work at a node is not constant: it is the two height computations, which cost $\Theta(n)$ at a node holding n nodes. So

$$T(n) = 2T(n/2) + \Theta(n), \tag{8.13}$$

and Equation (8.13) is $a = 2$, $b = 2$, $f(n) = \Theta(n) = \Theta(n^{\log_b a})$ — **case 2** — giving

$$T(n) = \Theta(n^{\log_b a}\lg n) = \Theta(n\lg n),$$

in agreement with the exact count Equation (8.10) derived in §8.9.3 and measured in **Table 8.6**. Two independent routes, one answer: that is how you know an analysis is right.

8.9.7 Common wrong turns on this problem

1. **Checking only the root.** `return abs(bf(t)) <= 1;`. The definition says *every* node. This is the exact analog of the local check of Problem 8.3. The counterexample is small: the tree $10\{5\{3\{1,.\},.\},20\}$ is balanced at the root and not at node 5.
2. **Returning the height where a yes-or-no answer was asked for**, or vice versa. Keep the sentinel routine `private` and let the public `isHeightBalanced` do the comparison with -2, exactly as **Listing 8.7** does.
3. **Choosing -1 as the failure flag.** -1 is the height of the empty tree. Using it makes every empty subtree look unbalanced, and the code then reports that the single-node tree is not an AVL tree. Test the single node; it must say yes.
4. **Forgetting `abs`.** Writing `hr - hl > 1` catches right-heavy faults and misses left-heavy ones. The counterexample of **Figure 8.4** is left-heavy, so this bug fails on the figure's own tree.
5. **Trusting a cached height field instead of computing it.** If the tree has been built by hand or by a buggy routine, its stored heights may be wrong, and a checker that reads them is checking nothing. That is why **Listing 8.7** deliberately declares its node type *without* a height field. §8.19 shows what happens in production when a validator trusts a stored field.

8.10 Problem 8.7 — `insertHB`: Insertion That Keeps the Tree Balanced

8.10.1 The problem

An AVL tree is a *height-balanced* (HB) tree. Write a recursive function `MyTree insertHB(int n, MyTree t)` which inserts `n` into `t` and returns the resulting tree, preserving the AVL condition.

Hint: `insertHB` should use two helper functions, `rebalanceForLeft` and `rebalanceForRight` (Problems 8.8 and 8.9). The same two helpers will serve deletion, `deleteHB`, in Problem 8.10. Their contract is:

`rebalanceForLeft(t)` is called when the left subtree of `t` may have grown taller by one notch. If it is indeed taller than the right subtree by two notches, it returns a height-balanced version of `t`, using single or double rotations. The subtrees of `t` are assumed to be already height-balanced, and no effort is made to rebalance them. `rebalanceForRight(t)` does the same for the right subtree.

Problems 8.7, 8.8 and 8.9 are one design and should be read together. Read the last sentence of the contract twice: *the subtrees of `t` are assumed to be already height-balanced, and no effort is made to rebalance them.*

8.10.2 The contract, stated precisely

The hint gives `rebalanceForLeft` a contract, and the contract is the real content of the problem. Written out precisely:

> **CONTRACT (`rebalanceForLeft(t)`). Precondition.** t is non-empty. Both subtrees of t are themselves height-balanced. The left subtree *may* have just grown taller by one notch, so $\mathrm{bf}(t) \in \{-2,-1,0,+1\}$ — everything except $+2$. **Postcondition.** The return value is a height-balanced tree containing exactly the same values as t, in the same sorted order. **Cost.** $O(1)$: at most two rotations, each a constant number of `node` constructions. No subtree is traversed. **`rebalanceForRight(t)` is the mirror image**, with $\mathrm{bf}(t) \in \{-1,0,+1,+2\}$.

The phrase "no effort is made to rebalance them" is the license that makes the $O(1)$ cost possible. The function is allowed to *assume* its children are already fine, so it only has to fix the one node in front of it.

8.10.3 Deriving `insertHB`

Step 1 — start from ordinary BST insertion. Chapter 6 §6.10 gave it, and **Figure 8.5** shows it: descend, comparing, until you reach an empty slot, and put the new node there as a leaf.

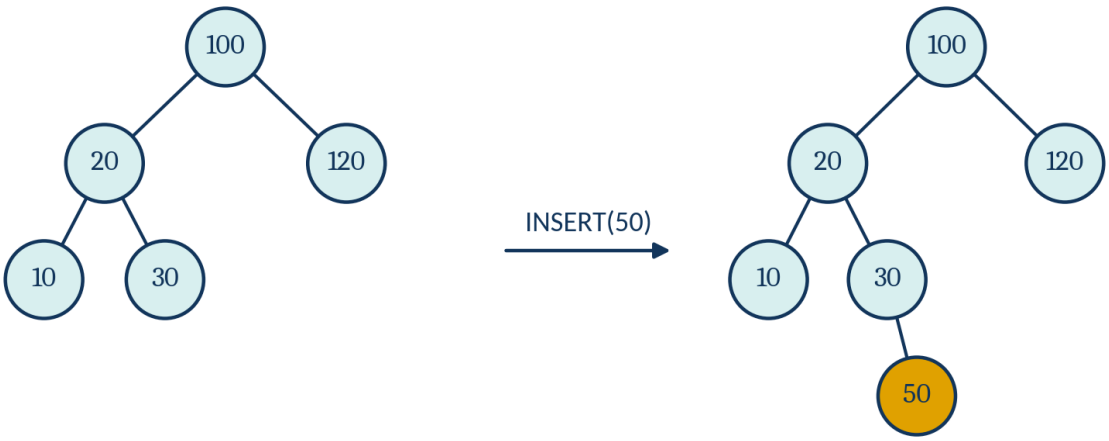


Figure 8.5 Insertion into a plain BST: a new element is always inserted as a leaf. The descent is the search, and the search is the placement.

In the functional style of **Listing 8.3**, "put it there" becomes "rebuild the path":

$$\text{insert}(n, \varnothing) = (n, \varnothing, \varnothing)$$

$$\text{insert}(n, (v, L, R)) = \begin{cases} (v, \text{insert}(n, L), R) & n < v \\ (v, L, \text{insert}(n, R)) & n > v \\ (v, L, R) & n = v \quad \text{(already present: no c} \end{cases}$$

Step 2 — ask what can go wrong. The new leaf has height 0, and the subtree it was inserted into may therefore be one taller than before. Every ancestor on the path from the new leaf to the root sees one of its two subtrees possibly one notch taller. Nothing else in the tree changed.

Step 3 — repair exactly those nodes, on the way back up. After `insert(n, L)` returns, the node v is being rebuilt, and its *left* subtree may have grown. That is precisely the precondition of `rebalanceForLeft`. Likewise on the right.

$$\text{insertHB}(n, \varnothing) = (n, \varnothing, \varnothing)$$

$$\text{insertHB}(n, (v, L, R)) = \begin{cases} \text{rebalanceForLeft}(v, \text{insertHB}(n, L), R) & n < \\ \text{rebalanceForRight}(v, L, \text{insertHB}(n, R)) & n > \\ (v, L, R) & n = \end{cases} \tag{8.14}$$

That is the solution, Equation (8.14), and it is nine lines of code. Notice how little there is to it once the rebalancers are assumed: `insertHB` is ordinary BST insertion with one function call wrapped around each rebuilt node.

Step 4 — why does this preserve the invariant? By induction on the recursion, from the bottom up. The base case returns a single leaf, which is height-balanced. At an internal node, the recursive call returns a height-balanced subtree (inductive hypothesis) whose height is either unchanged or one greater than before. The other subtree is untouched and was balanced. So the precondition of the rebalancer holds, and by the rebalancer's postcondition the node returns balanced. The root therefore returns balanced.

Step 5 — the cost. The recursion follows one root-to-leaf path: at most $h + 1$ levels. Each level does $\Theta(1)$ work plus one $O(1)$ rebalance. So insertion is $O(h) = O(\lg n)$ on an AVL tree. The number of *rotations* is at most 2 for the whole insertion — a fact Chapter 6 §6.20.2 proved, and which the output below exhibits.

> **WHY THE SIGNATURE RETURNS A `MyTree`.** Anyone meeting `insertHB(int n, MyTree t)` for the first time may ask why it does not simply modify `t` and return `void`. Because a rebalance can change which node is the *root* of the subtree. The caller must be told the new root, and returning it is the cleanest way. This is the same reason `rebalanceForLeft` returns a `MyTree`: it may hand back a different node from the one it was given.

8.10.4 A first look at the machine in action

Figure 8.6 works one insertion by hand: inserting 14 into the eight-node tree of §8.5. No rebalancing is needed, because every balance factor stays inside $\{-1, 0, +1\}$ — as the arithmetic of Chapter 6 §6.18.1 showed and as the program below confirms.

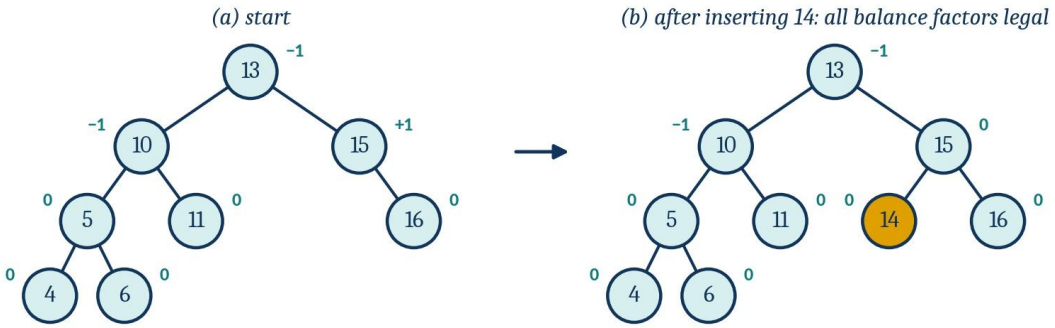


Figure 8.6 Inserting 14 into the eight-node tree. The insertion happens, the balance factors are recomputed up the path, and nothing leaves the legal range — so `insertHB` performs zero rotations. This is the common case.

8.11 Problem 8.8 — `rebalanceForLeft`: the Cases LL and LR

8.11.1 The problem

Write `MyTree rebalanceForLeft(MyTree t)`, the first helper of Problem 8.7. Its contract, repeated from §8.10.1:

`rebalanceForLeft(t)` is called when the left subtree of `t` may have grown taller by one notch. If it is indeed taller than the right subtree by two notches, it returns a height-balanced version of `t`, using single or double rotations. The subtrees of `t` are assumed to be already height-balanced, and no effort is made to rebalance them.

8.11.2 The one tool: rotation

A rotation is the only operation that changes a tree's shape without changing its inorder sequence. **Figure 8.7** is the picture, and it carries the proof underneath it.

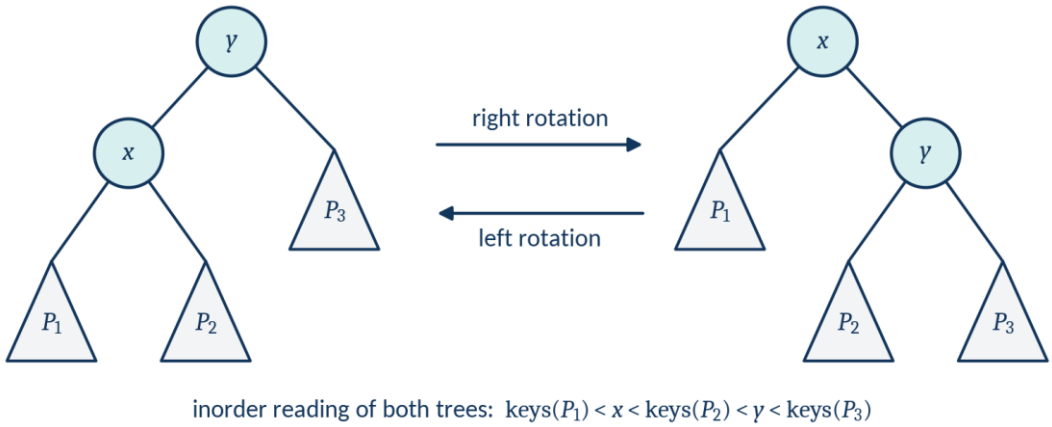


Figure 8.7 The two rotations. In both pictures the inorder reading is P_1, x, P_2, y, P_3, as the line beneath the figure states. A right rotation turns the left picture into the right one; a left rotation is its exact inverse.

Read the order relation stated beneath the figure: $\text{elements}(P_1) < x < \text{elements}(P_2) < y < \text{elements}(P_3)$. Both trees in **Figure 8.7** have that same inorder reading, so both are BSTs holding the same values. Only the shape differs.

In the functional style, a rotation is not a sequence of pointer assignments at all; it is one expression. A **right rotation** about t, which lifts t's left child, is

$$\text{rotateRight}((y, (x, P_1, P_2), P_3)) = (x, P_1, (y, P_2, P_3)), \tag{8.15}$$

which in code is

```
MyTree rotateRight(MyTree t) {
    MyTree l = leftOf(t);
    return node(value(l), leftOf(l), node(value(t), rightOf(l), rightOf(t)));
}
```

Read it against Equation (8.15). `value(l)` is x, the node that moves up. `leftOf(l)` is P_1, which stays with x. The inner `node(...)` is the new (y, P_2, P_3): the old root y, now holding P_2 on the left and its own old right subtree P_3 on the right. The heights are fixed automatically, because `node` computes each parent's height after its children exist — the inner call runs before the outer one. The ordering bug that Chapter 6 §6.16.1 warns about cannot occur.

A **left rotation** is the mirror: $\text{rotateLeft}((x, P_1, (y, P_2, P_3))) = ((y, (x, P_1, P_2), P_3))$.

Each rotation lifts one subtree by a level and lowers another by a level, and costs $\Theta(1)$: three `node` constructions at most, no traversal.

8.11.3 Case LL — one right rotation

Suppose $\text{bf}(t) = -2$: the left subtree is two notches taller. Let L be the left subtree. By the precondition, L is balanced, so $\text{bf}(L) \in \{-1, 0, +1\}$.

Case LL is $\text{bf}(L) \leq 0$ — the left subtree is itself left-heavy or level. **Figure 8.8** is the picture.

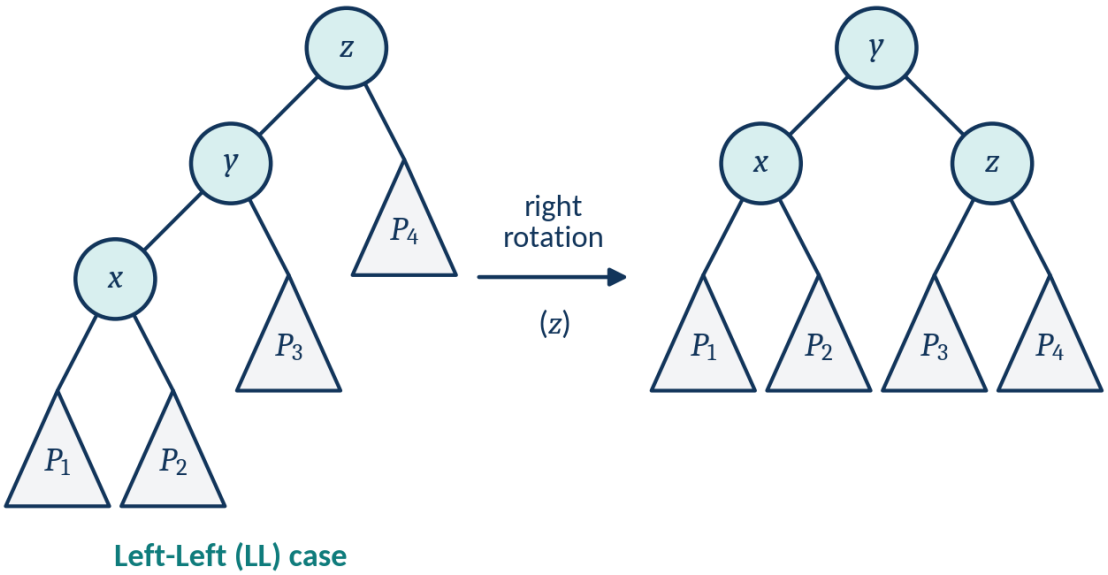


Figure 8.8 The Left-Left case: z is the unbalanced node, y its left child, x its left grandchild. One right rotation about z lifts y to the top. P_1 to P_4 are the four subtrees.

Do the height arithmetic. Write k for the height of the right subtree P_4 of z. Then $h(y) = k + 2$, because $\text{bf}(z) = -2$. Since $\text{bf}(y) \leq 0$, the taller of y's children is its left child x, so $h(x) = k + 1$ and $h(P_3) \in \{k, k + 1\}$; with $\text{bf}(y) \leq 0$ we get $h(P_3) \leq h(x)$, and since $h(y) = 1 + \max$ we must have $h(x) = k + 1$.

After one right rotation about z: y is on top; its left child is x with height $k + 1$; its right child is the new node z, which holds P_3 (height k or $k + 1$) and P_4 (height k).

- If $h(P_3) = k$: the new z has height $k + 1$, and $\text{bf}(z) = k - k = 0$. Then $\text{bf}(y) = (k + 1) - (k + 1) = 0$. Both legal. ✓
- If $h(P_3) = k + 1$ — which happens only when $\text{bf}(y) = 0$: the new z has height $k + 2$, and $\text{bf}(z) = k - (k + 1) = -1$. Then $\text{bf}(y) = (k + 2) - (k + 1) = +1$. Both legal. ✓

So one right rotation suffices in both sub-cases, and that is why the test is $\text{bf}(L) \leq 0$ and not $\text{bf}(L) < 0$. §8.11.5 returns to this, because it is the single subtlest point in Problems 8.7 to 8.10.

8.11.4 Case LR — a left rotation, then a right rotation

Case LR is $\text{bf}(L) > 0$ — the left subtree is right-heavy, so the path from the root down into the tall part zig-zags: left, then right. **Figure 8.9** shows both steps.

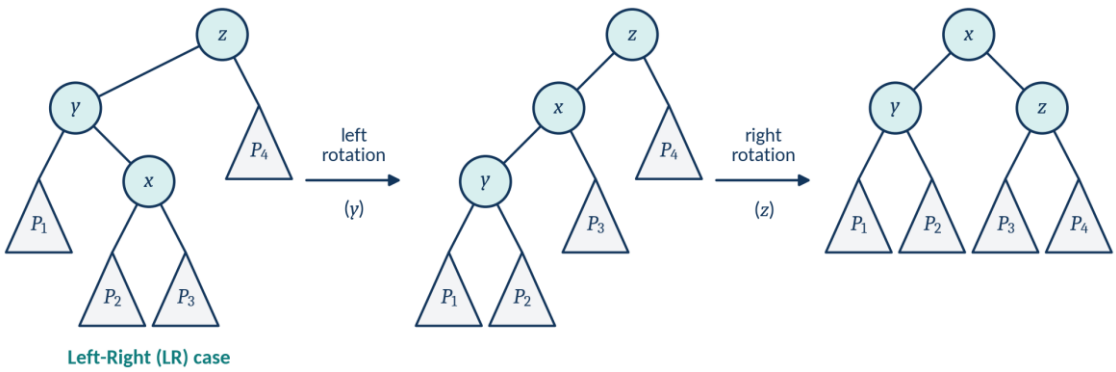


Figure 8.9 The Left-Right case. A single right rotation about z would leave the tree just as unbalanced the other way. First a left rotation about y straightens the zig-zag into a left-left lean; then one right rotation about z finishes the job.

Why does one rotation not work here? Because a right rotation about z lifts y's *left* subtree, and in the LR case the tall part is y's *right* subtree, which the rotation does not lift at all — it merely moves it across. Try it on the smallest case: the three-node tree with root 10, left child 5, and 5's right child 7. Here $\mathrm{bf}(10) = -2$ and $\mathrm{bf}(5) = +1$. A right rotation about 10 gives root 5, left child empty, right child 10 whose left child is 7: now $\mathrm{bf}(5) = +2$. The imbalance has changed sides, not gone away.

The repair: first rotate **left** about L, which turns the zig-zag into a straight left lean; then rotate **right** about t. On the three-node example: left rotation about 5 gives the chain $10 \to 7 \to 5$ leaning left; then a right rotation about 10 gives root 7 with children 5 and 10, perfectly balanced.

So:

$$\mathrm{rebalanceForLeft}(t) = \begin{cases} t & \mathrm{bf}(t) \geq -1 \quad (\text{notl} \\ \mathrm{rotateRight}(t) & \mathrm{bf}(t) = -2 \text{ and bf} \\ \mathrm{rotateRight}(v, \mathrm{rotateLeft}(L), R) & \mathrm{bf}(t) = -2 \text{ and bf} \end{cases} \tag{8.16}$$

and in code Equation (8.16) is six lines:

```
MyTree rebalanceForLeft(MyTree t) {
    if (balanceFactor(t) >= -1) return t;              // -1, 0 or +1: legal already
    MyTree l = leftOf(t);                              // bf(t) == -2, so l exists
    if (balanceFactor(l) <= 0) return rotateRight(t);  // LL: one rotation
    // LR: straighten the zig-zag first, then rotate right
    return rotateRight(node(value(t), rotateLeft(l), rightOf(t)));
}
```

8.11.5 The subtlest point: why $\leq$ and not $<$

The test that separates LL from LR is `balanceFactor(l) <= 0`. Why include 0 on the LL side?

During insertion, $\mathrm{bf}(L) = 0$ **cannot happen.** If the left subtree grew by one notch and the node became unbalanced, then the child that grew is strictly taller than its sibling, so $\mathrm{bf}(L) \neq 0$. So for insertion alone (Problem 8.7), either `<= 0` or `< 0` would work.

During deletion, $\mathrm{bf}(L) = 0$ **absolutely can happen**, and then the correct repair is the *single* rotation. Here is the smallest instance. Take the tree

$$z = 10, \quad L = 5\{3,7\}, \quad R = 20,$$

so $h(L) = 1$, $h(R) = 0$, $\mathrm{bf}(10) = -1$: balanced. Now delete 20. The right subtree becomes empty with height -1, so $\mathrm{bf}(10) = -1 - 1 = -2$, and $\mathrm{bf}(L) = \mathrm{bf}(5) = 0 - 0 = 0$.

- With `<= 0` we take LL and rotate right about 10: the result is root 5, left child 3, right child 10 holding 7 on the left. Heights: $h(3) = 0$, $h(10) = 1$, $h(5) = 2$. Balance factors: $\mathrm{bf}(10) = -1$, $\mathrm{bf}(5) = 1 - 0 = +1$. **Balanced.** ✓
- With `< 0` we would take LR: rotate left about 5, giving the left subtree $7\{5\{3,.\},.\}$ of height 2 — one *taller* than before — then rotate right about 10, giving root 7 with left child $5\{3,.\}$ of height 1 and right child 10 of height 0. That happens to be balanced here, but the double rotation has done unnecessary work and, in the general case where P_2 and P_3 have equal non-zero heights, produces $\mathrm{bf} = \pm 2$ at the new root. **Wrong in general.**

> **THE $\leq$ IS NOT A TYPO AND IT IS NOT DECORATION.** The hint of Problem 8.7 says the same two helpers will serve deletion. That single sentence is what forces `<= 0`. A version written with `< 0` passes every insertion test and fails on deletion, which is exactly the kind of bug that

is hardest to find late at night. If you are asked one question about `rebalanceForLeft`, it will be this one.

8.12 Problem 8.9 — `rebalanceForRight`: the Cases RR and RL

8.12.1 The problem

Write `MyTree rebalanceForRight(MyTree t)`, the second helper of Problem 8.7: the same contract as `rebalanceForLeft`, with the roles of the two subtrees exchanged. It is called when the *right* subtree of `t` may have grown taller by one notch.

8.12.2 The mirror principle

This is the cheapest problem in the chapter once Problem 8.8 is solved, and the reason is worth stating as a principle.

THE MIRROR PRINCIPLE. Reflecting a binary search tree left-to-right maps: left ↔ right, $<$ ↔ $>$, $\leq$ ↔ $\geq$, bf ↔ −bf, rotateRight ↔ rotateLeft, minimum ↔ maximum, ascending ↔ descending. Any correct routine, reflected symbol by symbol, is a correct routine for the mirrored problem. So once Problem 8.8 is right, Problem 8.9 is a transcription — and once `max` is right, `min` is free.

Apply the mirror to §8.11.4 line by line.

- `if (balanceFactor(t) >= -1) return t;` becomes `if (balanceFactor(t) <= 1) return t;`
- `MyTree l = leftOf(t);` becomes `MyTree r = rightOf(t);`
- `if (balanceFactor(l) <= 0) return rotateRight(t);` becomes `if (balanceFactor(r) >= 0) return rotateLeft(t);`
- `return rotateRight(node(value(t), rotateLeft(l), rightOf(t)));` becomes `return rotateLeft(node(value(t), leftOf(t), rotateRight(r)));`

Six lines, mechanically obtained:

```
MyTree rebalanceForRight(MyTree t) {
    if (balanceFactor(t) <= 1) return t;                // legal already
    MyTree r = rightOf(t);                              // bf(t) == +2, so r exists
    if (balanceFactor(r) >= 0) return rotateLeft(t);    // RR: one rotation
    // RL: straighten, then rotate left
    return rotateLeft(node(value(t), leftOf(t), rotateRight(r)));
}
```

8.12.3 Cases RR and RL, in pictures

Figure 8.10 and **Figure 8.11** show the two cases.

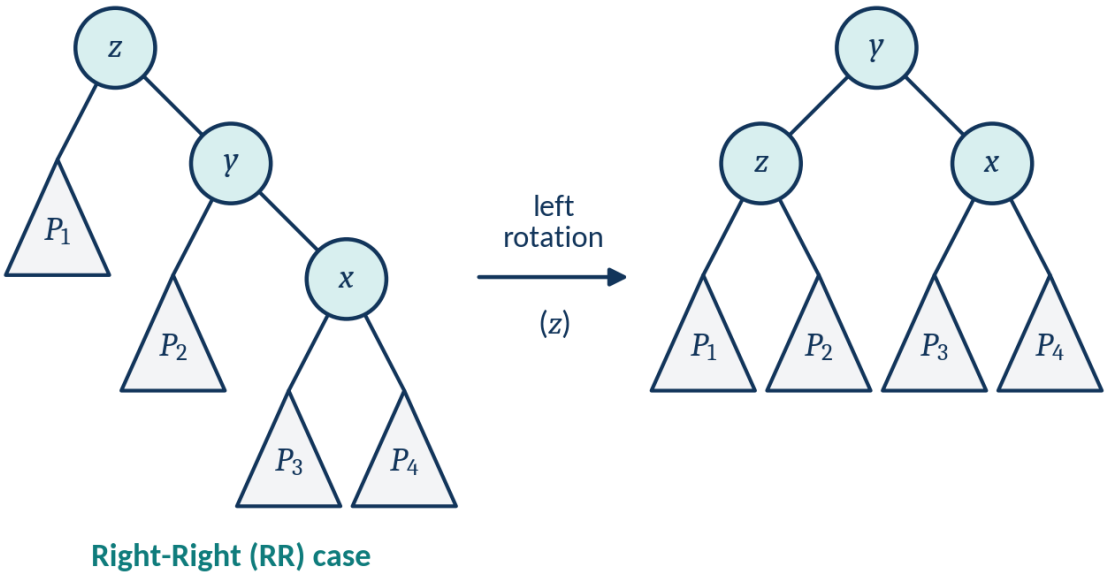


Figure 8.10 The Right-Right case: y is the right child of z and x is the right child of y. One left rotation about z repairs it. This is the exact mirror of **Figure 8.8**.

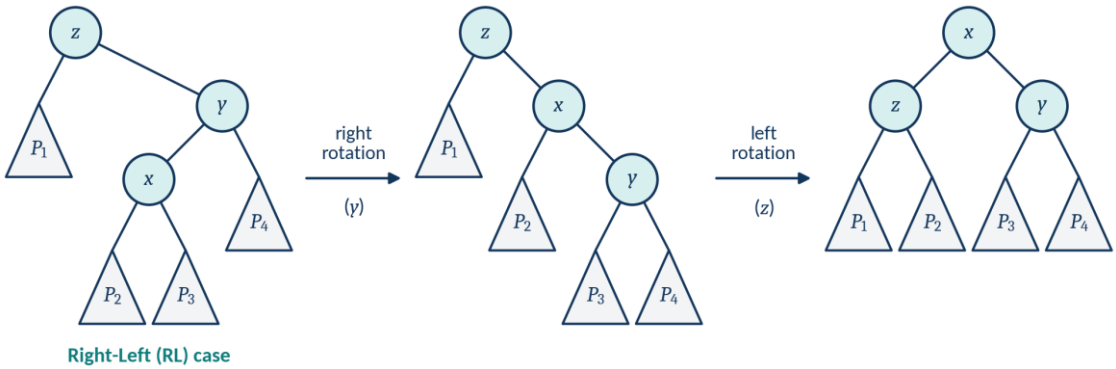


Figure 8.11 The Right-Left case: the zig-zag is right then left. A right rotation about y straightens it, then a left rotation about z finishes. The exact mirror of **Figure 8.9**.

Table 8.7 is the whole decision table for Problems 8.8 and 8.9 on one page. It is the thing to memorize, and it is four rows.

Table 8.7 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	$\text{bf}(L) \leq 0$	**LL**	one `rotateRight(t)`
-2	$\text{bf}(L) > 0$	**LR**	`rotateLeft(L)`, then `rotateRight(t)`
$+2$	$\text{bf}(R) \geq 0$	**RR**	one `rotateLeft(t)`
$+2$	$\text{bf}(R) < 0$	**RL**	`rotateRight(R)`, then `rotateLeft(t)`

8.12.4 Common wrong turns on Problems 8.8 and 8.9

1. **Getting the outer guard backward.** Writing `if (balanceFactor(t) <= -1) return t;` in `rebalanceForLeft` returns early on a tree with bf $= -2$, so nothing is ever repaired. Test: build the chain 30,20,10 by insertion and check the result is 20{10,30}.
2. **Using `<` instead of `<=`** on the case test. §8.11.5. Passes insertion, fails deletion.

3. **Rotating about the wrong node in the double case.** The first rotation is about the *child*, not the node itself. Writing `rotateLeft(t)` then `rotateRight(t)` is a no-op composition: the two cancel.
4. **Copying `rebalanceForLeft` into `rebalanceForRight` and changing only some of the symbols.** The mirror has *four* changes; it is easy to make only three. Check each one against the list in §8.12.2.
5. **Forgetting that the rebalancer may legitimately do nothing.** Both functions are called on *every* node of the insertion path, and most of those calls return `t` unchanged. A version that assumes it is always called on an unbalanced node will dereference an empty subtree.
6. **Exposing the helpers.** `rebalanceForLeft` and `rebalanceForRight` exist to serve `insertHB` and `deleteHB`; a caller that invokes them outside that contract gets no guarantee.

8.13 Problem 8.10 — `deleteHB`: Deletion That Keeps the Tree Balanced

8.13.1 The problem

Write a recursive function `MyTree deleteHB(MyTree t, int x)` which deletes `x` from `t` and returns the resulting tree, preserving the AVL condition. It must use the two helpers of Problems 8.8 and 8.9.

8.13.2 The three deletion cases

Chapter 6 gave the three cases of ordinary BST deletion; **Figure 8.12**, **Figure 8.13** and **Figure 8.14** show them.

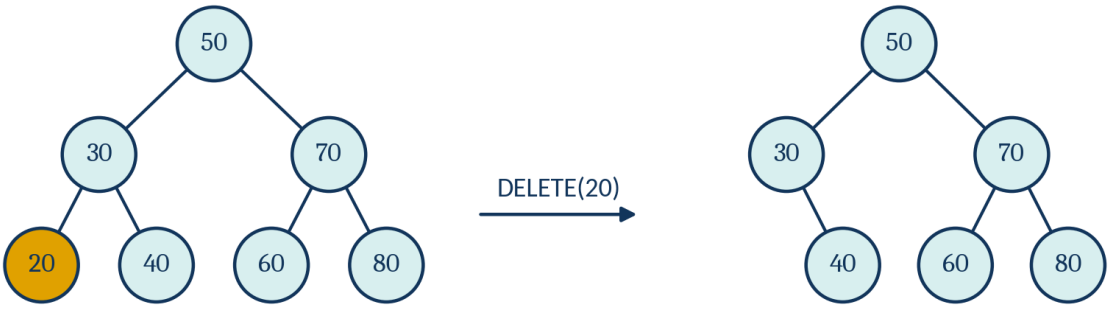


Figure 8.12 The first deletion case: the node to be deleted is a leaf. Simply remove it.

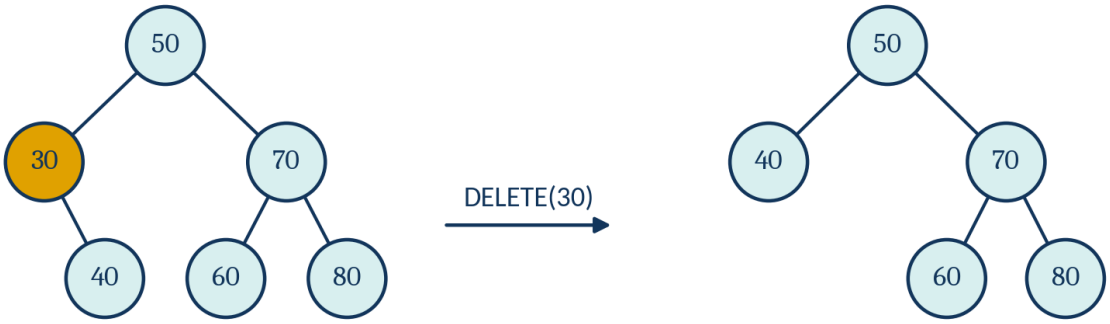


Figure 8.13 The second case: the node has exactly one child. "Copy the child to the node will be deleted, then delete the child" — equivalently, in the functional style, return the child in place of the node.

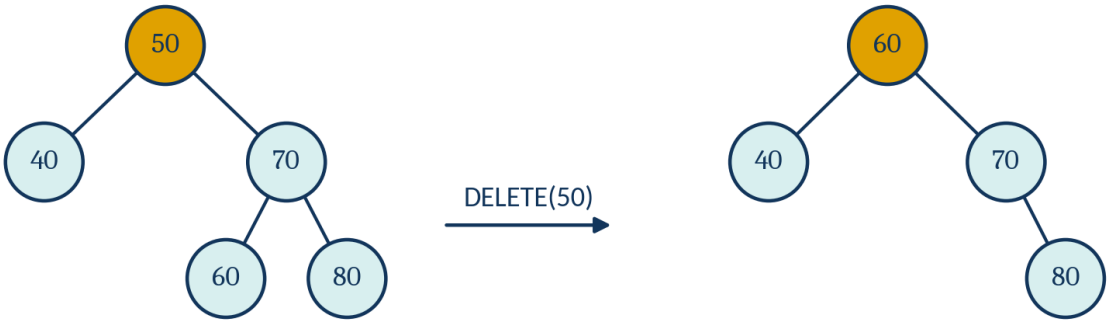


Figure 8.14 The third case: the node has two children. Find its inorder successor (or, symmetrically, its predecessor), copy that value into the node, and delete the successor from the right subtree — a smaller problem of case 1 or case 2.

In the functional style the three cases collapse pleasantly, because "return the child in place of the node" handles cases 1 and 2 together: a leaf's left subtree is empty, so returning the right subtree returns the empty tree.

$$\text{del}((v, L, R)) \text{ when } v \text{ is the target} = \begin{cases} R & L = \emptyset \quad \text{(cases 1} \\ L & R = \emptyset \quad \text{(case 2,} \\ (m, \text{del}(L, m), R) & \text{otherwise, wher} \end{cases}$$

We use the inorder **predecessor** $m = \max(L)$ rather than the successor $\min(R)$. Both are correct; the predecessor is the maximum of the left subtree, which Problem 8.5 already gave us for free, and it keeps the rebalance call on the same side throughout. §8.13.5 discusses the choice.

8.13.3 The trap: which rebalancer to call

Here is the single most common serious error in deletion, and it is a one-word error.

When the target is **smaller** than the current node, we recurse into the **left** subtree. That subtree may get *shorter*. A shorter left subtree means the **right** subtree may now be two notches taller. So the node must be repaired with `rebalanceForRight`.

Read that again, because it is the opposite of `insertHB`. In insertion, recursing left may make the left *taller*, so we call `rebalanceForLeft`. In deletion, recursing left may make the left *shorter*, so we call `rebalanceForRight`.

> **INSERT LEFT ⇒ `rebalanceForLeft`. DELETE LEFT ⇒ `rebalanceForRight`.** Write those two lines down before you write the function. The rebalancer is named after *the side that may have grown*, and deleting from a side makes the **other** side relatively taller. Getting this backward produces code that compiles, passes the insertion tests, and quietly leaves unbalanced trees behind — the worst kind of bug, because nothing crashes.

So:

$$\text{deleteHB}(\varnothing, x) = \varnothing$$

$$\text{deleteHB}((v, L, R), x) = \begin{cases} \text{rebalanceForRight}(v, \text{deleteHB}(L, x), R) & x < \\ \text{rebalanceForLeft}(v, L, \text{deleteHB}(R, x)) & x > \\ \text{the three cases of §8.13.2} & x = \end{cases} \tag{8.17}$$

Compare Equation (8.17) with Equation (8.14): the structure is the same and the rebalancers are exchanged. In the two-children case, after replacing v by $m = \max(L)$ and deleting m from L, the left subtree has again become shorter, so that node too is wrapped in `rebalanceForRight`.

8.13.4 Why deletion may need many rebalancings, and insertion at most one

Chapter 6 §6.20.2 established the asymmetry; here is the reason in two sentences, because it is one of the most instructive facts about AVL trees.

Insertion. A rebalance at the lowest unbalanced node restores that subtree to the height it had *before* the insertion. So every ancestor above it sees no change at all, and no further repair is possible or needed. At most one rebalancing — at most two rotations.

Deletion. A rebalance at the lowest unbalanced node may leave that subtree *one shorter* than before. That shortening is itself a change, which may unbalance the parent, which may shorten again, and so on all the way to the root. Up to $h = \Theta(\lg n)$ rebalancings.

Exercise 7 constructs the worst case explicitly — a minimum-node AVL tree, the *Fibonacci tree* of Chapter 6 §6.15.1 — and **Listing 8.14** in §8.23 measures the cascade: deleting one key from the minimal AVL tree of height 12 performs six rotations, and the count grows as $\lfloor h/2 \rfloor$.

8.13.5 Predecessor or successor?

The problem does not say which to use, and both are correct. The trade-offs:

- **Predecessor, $m = \max(L)$.** Reuses the `max` of Problem 8.5 directly. The follow-up delete happens in the left subtree, so the node is repaired with `rebalanceForRight` — the *same* rebalancer as the $x < v$ branch, which makes the code more uniform and easier to get right. This is what **Listing 8.8** does.
- **Successor, $m = \min(R)$.** The classical textbook choice, matching **Figure 8.14** and Chapter 6 §6.11.3. The follow-up delete happens in the right subtree, so the repair is `rebalanceForLeft`.

Choosing one and saying which, in one sentence, is all that is needed. Choosing one and calling the *wrong* rebalancer afterward is the trap of §8.13.3 wearing a different hat.

The worked deletion of Chapter 6, shown in **Figure 8.15** and **Figure 8.16**, is case 1: the node 32 is a leaf. The interesting part is what happens afterward.

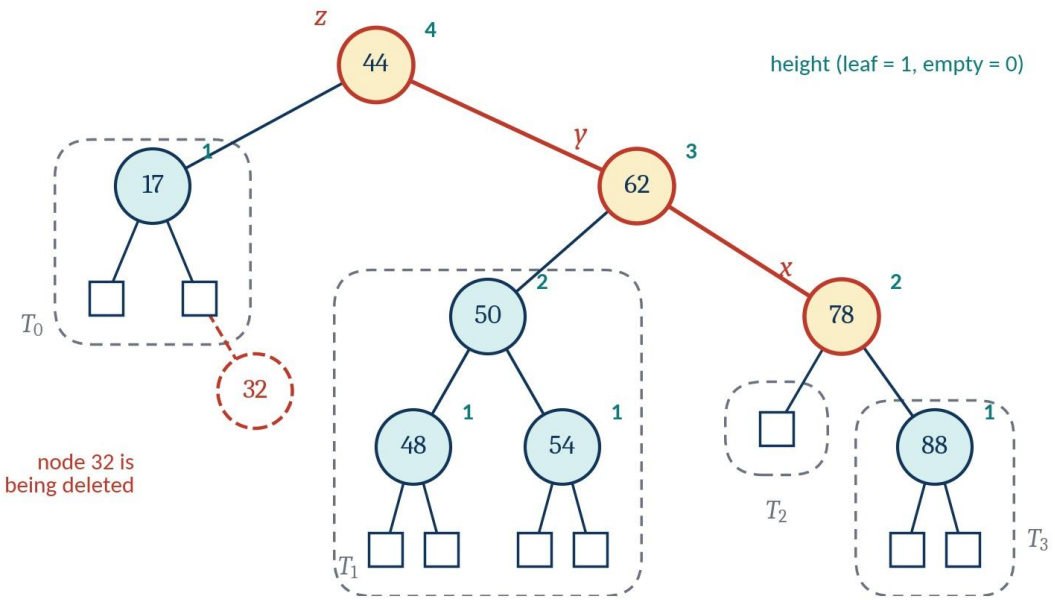


Figure 8.15 Before the deletion: the node 32 is about to be deleted, and the first unbalanced ancestor $z = 44$, its taller child $y = 62$ and y's taller child $x = 78$ are marked. Heights in the figure count nodes, so a leaf is 1 — one more than our convention everywhere.

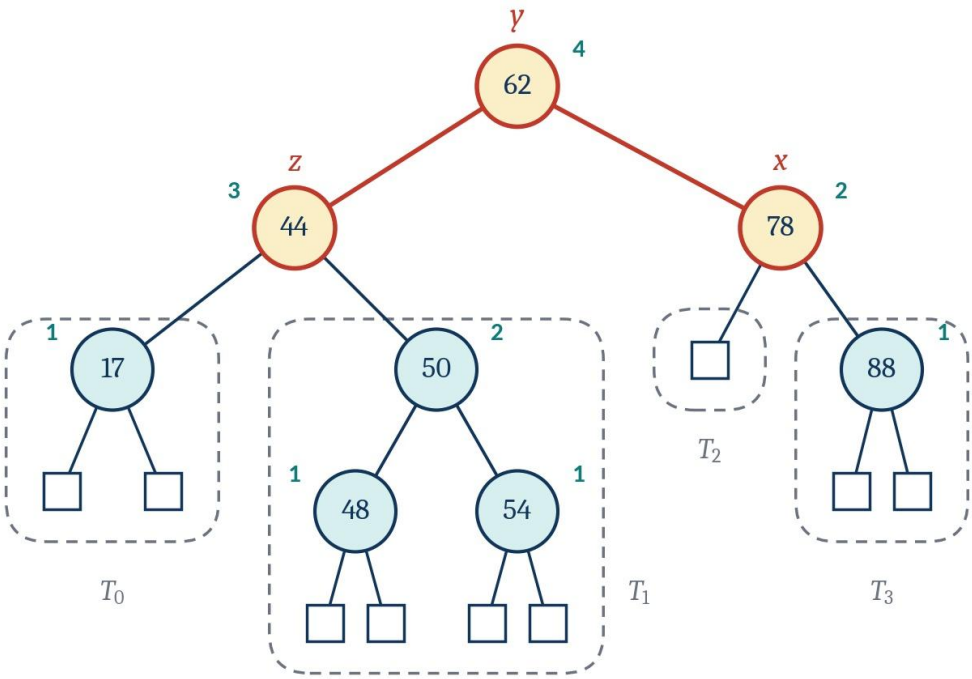


Figure 8.16 After one left rotation about 44. Node 62 is the new root, 44 has dropped to its left holding the subtree rooted at 50, and 78 is unchanged on the right.

In the vocabulary of Problem 8.10: deleting 32 from the right subtree of 17 makes 17's right side shorter, so 17 is repaired with `rebalanceForLeft` — which does nothing, since 17 is now a balanced leaf. Then 44's *left* side has become shorter, so 44 is repaired with `rebalanceForRight`, which finds bf(44)$= +2$ and bf(62) $= 0 \geq 0$, takes the RR row of **Table 8.7**, and performs one left rotation. That is exactly the conclusion **Figure 8.15** draws — the case is Right-Right, so a left rotation is done — reached by the decision table rather than by inspecting the picture.

8.13.6 Problems 8.7 to 8.10 as one program

Listing 8.8 is all four solutions in one complete program. Its `main` replays every one of the five insertion examples of Chapter 6 and its deletion example, then runs a thousand-element stress test that checks the AVL invariant, the sorted order and the theoretical height bound of Chapter 6.

Listing 8.8 Problems 8.7 to 8.10 in one program: `insertHB`, `rebalanceForLeft`, `rebalanceForRight`, `deleteHB`, plus an AVL self-check, a sorted-order check and a stress test against the theoretical height bound.

```
// avlhb.cpp -- Problems 8.7 to 8.10: insertHB(), rebalanceForLeft(),
//              rebalanceForRight() and deleteHB(), in functional style.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <cmath>
#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; }
int    value(MyTree t)      { return t->value; }
MyTree leftOf(MyTree t)     { return t->left; }
MyTree rightOf(MyTree t)    { return t->right; }

// The ONLY constructor. It recomputes the height from the two subtrees, so a
// tree built through node() can never carry 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;
}
int balanceFactor(MyTree t) { return height(rightOf(t)) - height(leftOf(t)); }

long long rotations = 0;    // counts single rotations performed

// ---------- the two rotations, in functional form ----------
MyTree rotateRight(MyTree t) {                  // t's left child moves up
    ++rotations;
    MyTree l = leftOf(t);
    return node(value(l), leftOf(l), node(value(t), rightOf(l), rightOf(t)));
}
MyTree rotateLeft(MyTree t) {                   // t's right child moves up
    ++rotations;
    MyTree r = rightOf(t);
    return node(value(r), node(value(t), leftOf(t), leftOf(r)), rightOf(r));
}
```

Listing 8.8 (Continued) *Problems 8.7 to 8.10 in one program: insertHB, rebalanceForLeft, rebalanceForRight, deleteHB, plus an AVL self-check, a sorted-order check and a stress test against the theoretical height bound.*

```
// ---------- Problem 8.8 ----------
// Called when the LEFT subtree of t may have grown taller by one notch.
MyTree rebalanceForLeft(MyTree t) {
    if (balanceFactor(t) >= -1) return t;               // -1, 0 or +1: legal already
    MyTree l = leftOf(t);                               // bf(t) == -2, so l exists
    if (balanceFactor(l) <= 0) return rotateRight(t);   // LL: one rotation
    // LR: straighten the zig-zag first, then rotate right
    return rotateRight(node(value(t), rotateLeft(l), rightOf(t)));
}

// ---------- Problem 8.9 ----------
// The exact mirror image: the RIGHT subtree may have grown taller by one notch.
MyTree rebalanceForRight(MyTree t) {
    if (balanceFactor(t) <= 1) return t;                // legal already
    MyTree r = rightOf(t);                              // bf(t) == +2, so r exists
    if (balanceFactor(r) >= 0) return rotateLeft(t);    // RR: one rotation
    // RL: straighten, then rotate left
    return rotateLeft(node(value(t), leftOf(t), rotateRight(r)));
}

// ---------- Problem 8.7 ----------
MyTree insertHB(int n, MyTree t) {
    if (isEmpty(t)) return node(n, emptyTree(), emptyTree());
    if (n < value(t))
        return rebalanceForLeft (node(value(t), insertHB(n, leftOf(t)), rightOf(t)));
    if (n > value(t))
        return rebalanceForRight(node(value(t), leftOf(t), insertHB(n, rightOf(t))));
    return t;                                           // duplicate: no change
}

// ---------- Problem 8.10 ----------
int maxOf(MyTree t) {                        // the O(h) max of Problem 8.5
    return isEmpty(rightOf(t)) ? value(t) : maxOf(rightOf(t));
}
MyTree deleteHB(MyTree t, int x) {
    if (isEmpty(t)) return t;                           // x is not in the tree
    if (x < value(t))
        // the LEFT side just got SHORTER, so the RIGHT side may be two taller
        return rebalanceForRight(node(value(t), deleteHB(leftOf(t), x), rightOf(t)));
    if (x > value(t))
        return rebalanceForLeft (node(value(t), leftOf(t), deleteHB(rightOf(t), x)));
    if (isEmpty(leftOf(t)))  return rightOf(t);         // 0 or 1 child
    if (isEmpty(rightOf(t))) return leftOf(t);
    int m = maxOf(leftOf(t));                           // inorder predecessor
    return rebalanceForRight(node(m, deleteHB(leftOf(t), m), rightOf(t)));
}
```

Listing 8.8 (Continued) *Problems 8.7 to 8.10 in one program: insertHB, rebalanceForLeft, rebalanceForRight, deleteHB, plus an AVL self-check, a sorted-order check and a stress test against the theoretical height bound.*

```
// ---------- checking ----------
string shape(MyTree t) {
    if (isEmpty(t)) return ".";
    string s = to_string(value(t)) + "(" + to_string(balanceFactor(t)) + ")";
    if (isEmpty(leftOf(t)) && isEmpty(rightOf(t))) return s;
    return s + "{" + shape(leftOf(t)) + "," + shape(rightOf(t)) + "}";
}
int  countNodes(MyTree t) { return isEmpty(t) ? 0 : 1 + countNodes(leftOf(t)) +
countNodes(rightOf(t)); }
bool isAVL(MyTree t) {
    if (isEmpty(t)) return true;
    if (t->h != 1 + max(height(leftOf(t)), height(rightOf(t)))) return false;  // stale
height
    if (abs(balanceFactor(t)) > 1) return false;
    return isAVL(leftOf(t)) && isAVL(rightOf(t));
}
void collect(MyTree t, vector<int>& v) {
    if (isEmpty(t)) return;
    collect(leftOf(t), v); v.push_back(value(t)); collect(rightOf(t), v);
}
bool isSorted(MyTree t) {
    vector<int> v; collect(t, v);
    return is_sorted(v.begin(), v.end()) && adjacent_find(v.begin(), v.end()) == v.end();
}

void step(const char* what, MyTree t) {
    cout << "  " << left << setw(12) << what << shape(t)
         << "   [h=" << height(t) << ", rot=" << rotations
         << ", AVL=" << (isAVL(t) && isSorted(t) ? "yes" : "NO") << "]\n";
}

int main() {
    cout << "--- building the eight-node tree of Chapter 6 with insertHB ---\n";
    rotations = 0;
    MyTree t = emptyTree();
    for (int v : {13, 10, 15, 5, 11, 16, 4, 6}) t = insertHB(v, t);
    step("built", t);

    cout << "--- insert 14, no rotation expected ---\n";
    rotations = 0; MyTree t14 = insertHB(14, t); step("insert 14", t14);

    cout << "--- insert 3, Left-Left expected ---\n";
    rotations = 0; MyTree t3 = insertHB(3, t); step("insert 3", t3);

    cout << "--- insert 7, Left-Right expected ---\n";
    rotations = 0; MyTree t7 = insertHB(7, t); step("insert 7", t7);
```

Listing 8.8 (Continued) *Problems 8.7 to 8.10 in one program: insertHB, rebalanceForLeft, rebalanceForRight, deleteHB, plus an AVL self-check, a sorted-order check and a stress test against the theoretical height bound.*

```
    cout << "--- root 30, left 5, right 35{32,40}; insert 45, Right-Right ---\n";
    rotations = 0;
    MyTree u = emptyTree();
    for (int v : {30, 5, 35, 32, 40}) u = insertHB(v, u);
    step("built", u);
    rotations = 0; step("insert 45", insertHB(45, u));

    cout << "--- insert 15 into 5{2{1,4{3,.}},7{6,9{.,16}}}, Right-Left ---\n";
    rotations = 0;
    MyTree w = emptyTree();
    for (int v : {5, 2, 7, 1, 4, 6, 9, 3, 16}) w = insertHB(v, w);
    step("built", w);
    rotations = 0; step("insert 15", insertHB(15, w));

    cout << "--- the deletion example of Chapter 6 ---\n";
    rotations = 0;
    MyTree d = emptyTree();
    for (int v : {44, 17, 62, 32, 50, 78, 48, 54, 88}) d = insertHB(v, d);
    step("built", d);
    rotations = 0; MyTree d2 = deleteHB(d, 32); step("delete 32", d2);
    rotations = 0; step("delete 62", deleteHB(d, 62));   // a node with two children
    rotations = 0; step("delete 99", deleteHB(d, 99));   // absent: no change

    cout << "--- stress test: insert 1..1000 in order, then delete them all ---\n";
    rotations = 0;
    MyTree s = emptyTree();
    for (int v = 1; v <= 1000; ++v) s = insertHB(v, s);
    int n = countNodes(s);
    double bound = 1.4405 * log2((double)n + 2.0) - 1.3277;
    cout << "  n = " << n << ", height = " << height(s)
         << ", AVL = " << (isAVL(s) && isSorted(s) ? "yes" : "NO")
         << ", rotations = " << rotations << "\n";
    cout << "  theoretical bound 1.4405*lg(n+2)-1.3277 = " << fixed << setprecision(3) <<
bound
         << "  ->  height <= " << (int)floor(bound) << ": "
         << (height(s) <= bound ? "satisfied" : "VIOLATED") << "\n";
    rotations = 0;
    for (int v = 1; v <= 1000; ++v) s = deleteHB(s, v);
    cout << "  after deleting all: nodes = " << countNodes(s)
         << ", isEmpty = " << (isEmpty(s) ? "true" : "false")
         << ", rotations = " << rotations << "\n";
    return 0;
}
```

Compiled and run, it prints:

```
--- building the eight-node tree of Chapter 6 with insertHB ---
  built       13(-1){10(-1){5(0){4(0),6(0)},11(0)},15(1){.,16(0)}}   [h=3, rot=0, AVL=yes]
--- insert 14, no rotation expected ---
  insert 14   13(-1){10(-1){5(0){4(0),6(0)},11(0)},15(0){14(0),16(0)}}   [h=3, rot=0,
AVL=yes]
--- insert 3, Left-Left expected ---
  insert 3    13(-1){5(0){4(-1){3(0),.},10(0){6(0),11(0)}},15(1){.,16(0)}}   [h=3, rot=1,
AVL=yes]
--- insert 7, Left-Right expected ---
  insert 7    13(-1){6(0){5(-1){4(0),.},10(0){7(0),11(0)}},15(1){.,16(0)}}   [h=3, rot=2,
AVL=yes]
--- root 30, left 5, right 35{32,40}; insert 45, Right-Right ---
  built       30(1){5(0),35(0){32(0),40(0)}}   [h=2, rot=0, AVL=yes]
  insert 45   35(0){30(0){5(0),32(0)},40(1){.,45(0)}}   [h=2, rot=1, AVL=yes]
--- insert 15 into 5{2{1,4{3,.}},7{6,9{.,16}}}, Right-Left ---
  built       5(0){2(1){1(0),4(-1){3(0),.}},7(1){6(0),9(1){.,16(0)}}}   [h=3, rot=0,
AVL=yes]
  insert 15   5(0){2(1){1(0),4(-1){3(0),.}},7(1){6(0),15(0){9(0),16(0)}}}   [h=3, rot=2,
AVL=yes]
--- the deletion example of Chapter 6 ---
  built       44(1){17(1){.,32(0)},62(0){50(0){48(0),54(0)},78(1){.,88(0)}}}   [h=3, rot=0,
AVL=yes]
  delete 32   62(-1){44(1){17(0),50(0){48(0),54(0)}},78(1){.,88(0)}}   [h=3, rot=1,
AVL=yes]
  delete 62   44(1){17(1){.,32(0)},54(0){50(-1){48(0),.},78(1){.,88(0)}}}   [h=3, rot=0,
AVL=yes]
  delete 99   44(1){17(1){.,32(0)},62(0){50(0){48(0),54(0)},78(1){.,88(0)}}}   [h=3, rot=0,
AVL=yes]
--- stress test: insert 1..1000 in order, then delete them all ---
  n = 1000, height = 9, AVL = yes, rotations = 990
  theoretical bound 1.4405*lg(n+2)-1.3277 = 13.032  ->  height <= 13: satisfied
  after deleting all: nodes = 0, isEmpty = true, rotations = 491
```

Check it against the hand calculations of Chapter 6, one line at a time.

- **Insert 14:** `rot=0`. **Figure 8.6** needs no rotation. ✓
- **Insert 3:** `rot=1`, and the resulting shape is root 13, left child 5 with children 4 (holding 3) and 10 (holding 6 and 11). Chapter 6 §6.18.2 derived exactly that shape by hand, with $\mathrm{bf}(4) = -1$, $\mathrm{bf}(10) = 0$, $\mathrm{bf}(5) = 0$, $\mathrm{bf}(13) = -1$ — and the printed shape shows `4(-1)`, `10(0)`, `5(0)`, `13(-1)`. ✓
- **Insert 7:** `rot=2`, the double rotation of the LR case, and the shape is root 13, left child 6 with children 5 (holding 4) and 10 (holding 7 and 11) — Chapter 6 §6.18.4's result exactly. ✓
- **Insert 45:** `rot=1`, and the shape is root 35, left child 30 with children 5 and 32, right child 40 with right child 45 — Chapter 6 §6.18.3. ✓
- **Insert 15:** `rot=2`, RL, and the right subtree becomes $7\{6,15\{9,16\}\}$ — Chapter 6 §6.18.5. ✓
- **Delete 32:** `rot=1`, and the shape is `62(-1){44(1){17(0),50(0){48(0),54(0)}},78(1){.,88(0)}}` — which is the tree of **Figure 8.16**, written on one line, and character-for-character the output Chapter 6 §6.20.3 quoted. ✓
- **Delete 62:** the two-children case. The predecessor of 62 in the left subtree is 54, and indeed 54 is the new root of that subtree; no rotation was needed. ✓
- **Delete 99:** a value not in the tree. The shape is unchanged and no rotation occurs — the base case `if (isEmpty(t)) return t;` doing its job. ✓
- **Stress test:** one thousand ascending insertions — the worst possible order for a plain BST, which would produce a chain of height 999 — give height 9. The theoretical bound is $1.4405\lg 1002 - 1.3277 = 13.032$, so the invariant holds with room to spare. All one thousand

deletions then empty the tree, with 491 rotations, and every intermediate tree passed both the AVL check and the sorted-order check.

> **A CORRECTNESS ARGUMENT IN ONE PARAGRAPH.** *"The four routines were checked against the six worked examples of Chapter 6. Inserting* 14 *requires no rotation; inserting* 3 *takes one right rotation, the LL case; inserting* 45 *takes one left rotation, RR; inserting* 7 *takes two rotations, LR; inserting* 15 *takes two rotations, RL; and deleting* 32 *takes one left rotation about* 44*, RR. The program's rotation counts are* 0,1,1,2,2,1 *respectively, and every resulting shape matches the hand-worked tree. Inserting* 1 *to* 1000 *in ascending order gives height* 9*, within the AVL bound* $1.4405\lg(n+2) - 1.3277 = 14.03$*."*
>
> That paragraph can be checked by anyone in thirty seconds, and it is worth more than any amount of prose about how the algorithm works.

8.13.7 Common wrong turns on this problem

1. **Calling the wrong rebalancer after a recursive delete.** §8.13.3. It is the error that produces silently unbalanced trees.
2. **Forgetting to rebalance after the two-children replacement.** The recursive `deleteHB(L, m)` shortens the left subtree, so that node needs `rebalanceForRight` too. It is easy to handle the two easy branches and forget the third.
3. **Deleting the successor's *node* instead of recursing.** Writing code that unlinks the successor by pointer surgery loses the rebalancing on the path down to it. Recurse; do not splice.
4. **Not handling "x is not in the tree".** The base case must return the tree unchanged, not crash and not delete something else. Test `deleteHB(t, 99)`.
5. **Handling only the leaf case** and returning `t` unchanged for nodes with children, so that the value is never actually removed. Always test by deleting the *root*.

8.14 Problems 8.11 to 8.15 — Red-Black Trees

The AVL tree is one way to keep a search tree short. Chapter 6 also met the other famous way, the **red-black tree**, which balances by *color* instead of by height difference. This section works five problems on red-black trees. Together with `isBST` — which carries over from Problem 8.3 unchanged, because a red-black tree is first of all a binary search tree — they make a complete validator for the structure, and then the two rotations that every red-black insertion and deletion is built from.

8.14.1 The problems, and how they relate to the AVL problems

Every problem in this section works on the same node type: a key, a `red` flag, `left`, `right` and `parent` pointers, and a single shared sentinel `NIL` that is black and stands for every empty subtree. **Table 8.8** lists the five problems.

Table 8.8 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 8.9 sets the red-black problems beside the AVL problems. They are recognizably the same family: a validation problem, two rotation problems, and structural checks in between.

Table 8.9 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	$\lvert\mathrm{bf}(x)\rvert \le 1$	colors, plus a uniform black-height
Style	functional: rebuild and return	imperative: mutate, with `parent` pointers

8.14.2 The five red-black properties

A red-black tree is a binary search tree in which every node carries one extra bit of color, red or black, and in which:

1. Every node is either red or black.
2. The root is black.
3. Every leaf — here, every `NIL` sentinel — is black.
4. If a node is red, both its children are black. (Equivalently: no red node has a red child.)
5. For each node, every simple path from that node down to a descendant leaf contains the same number of black nodes. That number, not counting the node itself, is the node's **black-height**.

Properties 4 and 5 together force the tree to be short. The argument, in one line: the longest root-to-leaf path can alternate red and black, and the shortest can be all black, so the longest is at most twice the shortest, giving $h \le 2\lg(n+1)$. Chapter 6 §6.21.1 compared this with the AVL bound

$1.4405\lg(n+2) - 1.3277$: red-black trees are allowed to be taller, and in exchange they rebalance less often [1].

8.14.3 Problem 8.11 — counting the black nodes

The problem. Write `int countBlackNodes(const RBTree& t)`, returning the number of black nodes of `t`, not counting the `NIL` sentinels.

The solution. A plain postorder count: 0 at `NIL`, otherwise 1 if this node is black, plus the counts of both subtrees. $\Theta(n)$, and there is no cleverer alternative — every node must be examined, because color is not deducible from position. The one decision to make explicit is that `NIL` sentinels are *not* counted, which is the right choice: they are structural, not data.

8.14.4 Problem 8.12 — no red node with a red child

The problem. Write `bool validateProperty4(const RBTree& t)`, returning `true` exactly when property 4 holds.

The solution. At every node: if this node is red, neither child may be red. The test looks at a node and its *children* — a local test — and this time the local test is correct, because property 4 genuinely is a statement about parent–child pairs. Contrast §8.6.3, where the local test was wrong because the BST property is not. Knowing *when* a local test suffices is the whole skill, and Problems 8.3 and 8.12, side by side, make exactly that point.

8.14.5 Problem 8.13 — a uniform black-height

The problem. Write `bool validateBlackHeight(const RBTree& t)`, returning `true` exactly when property 5 holds, in one pass.

The solution. The sentinel technique of §8.9.4, exactly. A helper `computeBlackHeight` returns the subtree's black-height, or -1 to mean "the property already fails somewhere below". That is legitimate because a real black-height is never negative. The helper returns -1 if either child returned -1, or if the two children's black-heights differ; otherwise it returns the child's black-height, plus one if this node is black. One pass, $\Theta(n)$.

8.14.6 Problem 8.14 — the complete validator

The problem. Write `bool isValidRBTree(const RBTree& t)`, returning `true` exactly when `t` is a binary search tree satisfying all five red-black properties.

The solution. The assembly. Property 1 is free — the color is a `bool`, so it cannot be anything else. Property 3 is free — `NIL` is constructed black and is shared. Property 2 is one test on the root. The search-tree property, property 4 and property 5 are Problems 8.3, 8.12 and 8.13. The routine is four lines long; its real content is being able to say *which properties the code does not need to check, and why*.

8.14.7 Problem 8.15 — rotations with parent pointers

The problem. Write `void leftRotate(RBTree& t, RBNode* x)`, which lifts x's right child y into x's place and makes x the left child of y, updating every `parent` pointer and, if x was the root, the root of `t`. Then write its mirror, `rightRotate`.

The solution. The classical imperative rotation of Cormen et al. [2, ch. 13], in four numbered steps: set y; move y's inner subtree to x; relink x's parent to y; put x below y. The `NIL` sentinel

earns its keep here: because `NIL` is a real node, the test `if (x->parent == NIL) t.root = y;` needs no null check. Compare the functional rotations of §8.11.2, which need no parent pointers at all because the new subtree root is *returned* to the caller — nine lines against sixteen, and no pointer to forget.

> **THE POINTER UPDATES EVERYONE FORGETS.** In the imperative rotation there are *six* pointers to fix, in three pairs: x.right and y.left.parent; y.parent and the parent's child link; y.left and x.parent. It is easy to set the downward pointers and forget the upward ones, producing a tree that prints correctly and walks upward into the wrong place. The tests in **Listing 8.9** check `10's parent pointer was fixed` for exactly this reason.

8.14.8 The five problems as one running program

Listing 8.9 contains all five solutions, the narrow `int`-bound version of `isBST` from §8.6.7 together with its repair, and a set of checks that print `PASS` or `FAIL`.

Listing 8.9 Problems 8.11 to 8.15 in one program: the red-black validators, the imperative rotations with parent pointers, a set of checks, and a demonstration of the `int`-bound pitfall of §8.6.7.

```
// rbtree.cpp -- Problems 8.11 to 8.15: red-black validators and rotations with
// parent pointers, together with isBST from Problem 8.3.
#include <iostream>
#include <string>
#include <vector>
#include <climits>
using namespace std;

// ---------------- RBNode ----------------
struct RBNode {
    int     key   = 0;
    bool    red   = false;                  // true = RED, false = BLACK
    RBNode *left = nullptr, *right = nullptr, *parent = nullptr;
    bool isRed()   const { return red; }
    bool isBlack() const { return !red; }
};
RBNode* makeNil() {                         // the shared sentinel: BLACK, self-linked
    RBNode* n = new RBNode;
    n->left = n->right = n->parent = n;
    return n;
}
RBNode* const NIL = makeNil();
RBNode* newNode(int key) {                  // a fresh node is always RED
    RBNode* n = new RBNode;
    n->key = key; n->red = true;
    n->left = n->right = n->parent = NIL;
    return n;
}
string label(RBNode* n) {
    return n == NIL ? "NIL:B" : to_string(n->key) + ":" + (n->red ? "R" : "B");
}

// ---------------- RBTree ----------------
struct RBTree {
    RBNode* root = NIL;
    void bstInsert(int key) {               // plain BST insert, no color fixup
        RBNode *z = newNode(key), *y = NIL, *x = root;
        while (x != NIL) {
            y = x;
            if      (z->key < x->key) x = x->left;
            else if (z->key > x->key) x = x->right;
            else return;                    // duplicate key: do nothing
        }
        z->parent = y;
        if      (y == NIL)        root = z;
        else if (z->key < y->key) y->left  = z;
```

Listing 8.9 (Continued) *Problems 8.11 to 8.15 in one program: the red-black validators, the imperative rotations with parent pointers, a set of checks, and a demonstration of the int-bound pitfall of §8.6.7.*

```
        else                        y->right = z;
    }
};
RBNode* findNode(RBNode* n, int key) {
    if (n == NIL || key == n->key) return n;
    return key < n->key ? findNode(n->left, key) : findNode(n->right, key);
}
RBTree buildForTest(const vector<int>& keys, const vector<bool>& reds) {
    RBTree t;
    for (int k : keys) t.bstInsert(k);
    for (size_t i = 0; i < keys.size(); ++i) findNode(t.root, keys[i])->red = reds[i];
    return t;
}
void printTree(RBNode* n, const string& pre, bool isLeft) {
    if (n == NIL) return;
    cout << pre << (isLeft ? "|-- " : "`-- ") << label(n) << "\n";
    printTree(n->left,  pre + (isLeft ? "|   " : "    "), true);
    printTree(n->right, pre + (isLeft ? "|   " : "    "), false);
}

// ---------------- validators ----------------
// isBST() of Problem 8.3, in the tempting narrow form of section 8.6.7: int bounds
// seeded with INT_MIN and INT_MAX. Kept here on purpose, to show the pitfall.
static bool isBST(RBNode* n, int lowerBound, int upperBound) {
    if (n == NIL) return true;
    if (n->key <= lowerBound || n->key >= upperBound) return false;
    return isBST(n->left, lowerBound, n->key) && isBST(n->right, n->key, upperBound);
}
bool isBST(const RBTree& t) { return t.root == NIL || isBST(t.root, INT_MIN, INT_MAX); }

// The repair: bounds one step wider than any int can be.
static bool isBSTFixed(RBNode* n, long long lo, long long hi) {
    if (n == NIL) return true;
    if (n->key <= lo || n->key >= hi) return false;
    return isBSTFixed(n->left, lo, n->key) && isBSTFixed(n->right, n->key, hi);
}
bool isBSTRepaired(const RBTree& t) {
    return isBSTFixed(t.root, (long long)INT_MIN - 1, (long long)INT_MAX + 1);
}
```

Listing 8.9 (Continued) *Problems 8.11 to 8.15 in one program: the red-black validators, the imperative rotations with parent pointers, a set of checks, and a demonstration of the int-bound pitfall of §8.6.7.*

```
// Problem 8.11: countBlackNodes(). NIL sentinels are not counted.
static int countBlack(RBNode* n) {
    if (n == NIL) return 0;
    return (n->isBlack() ? 1 : 0) + countBlack(n->left) + countBlack(n->right);
}
int countBlackNodes(const RBTree& t) { return countBlack(t.root); }

// Problem 8.12: validateProperty4(): no RED node has a RED child.
static bool prop4(RBNode* n) {
    if (n == NIL) return true;
    if (n->isRed() && (n->left->isRed() || n->right->isRed())) return false;
    return prop4(n->left) && prop4(n->right);
}
bool validateProperty4(const RBTree& t) { return prop4(t.root); }

// Problem 8.13: validateBlackHeight(). -1 is the failure flag, because a real
// black-height is never negative -- the sentinel technique of Problem 8.6 again.
int computeBlackHeight(RBNode* n) {
    if (n == NIL) return 0;
    int l = computeBlackHeight(n->left), r = computeBlackHeight(n->right);
    if (l == -1 || r == -1 || l != r) return -1;
    return l + (n->isBlack() ? 1 : 0);
}
bool validateBlackHeight(const RBTree& t) {
    return t.root == NIL || computeBlackHeight(t.root) != -1;
}

// Problem 8.14: isValidRBTree(): all five properties at once.
bool isValidRBTree(const RBTree& t) {
    if (t.root == NIL)   return true;   // property 1 is automatic: the flag is a bool
    if (t.root->isRed()) return false;  // property 2: the root is black
    return isBST(t) && validateProperty4(t) && validateBlackHeight(t);
}                                       // property 3 holds by NIL's construction

// ---------------- Problem 8.15: rotations ----------------
// leftRotate(t, x): x's right child y moves up; x becomes y's left child.
void leftRotate(RBTree& t, RBNode* x) {
    RBNode* y = x->right;                   // 1. set y
    x->right = y->left;                     // 2. y's left subtree becomes x's right
    if (y->left != NIL) y->left->parent = x;
    y->parent = x->parent;                  // 3. link x's parent to y
    if      (x->parent == NIL)     t.root = y;
    else if (x == x->parent->left) x->parent->left  = y;
    else                           x->parent->right = y;
    y->left = x;                            // 4. put x on y's left
    x->parent = y;
```

Listing 8.9 (Continued) *Problems 8.11 to 8.15 in one program: the red-black validators, the imperative rotations with parent pointers, a set of checks, and a demonstration of the int-bound pitfall of §8.6.7.*

```
}
// rightRotate(t, y): the exact mirror image.
void rightRotate(RBTree& t, RBNode* y) {
    RBNode* x = y->left;
    y->left = x->right;
    if (x->right != NIL) x->right->parent = y;
    x->parent = y->parent;
    if      (y->parent == NIL)     t.root = x;
    else if (y == y->parent->left) y->parent->left  = x;
    else                           y->parent->right = x;
    x->right = y;
    y->parent = x;
}

// ---------------- checks ----------------
int passed = 0, failed = 0;
void check(const string& name, bool cond) {
    (cond ? passed : failed)++;
    cout << (cond ? "  PASS: " : "  FAIL: ") << name << "\n";
}

int main() {
    RBTree valid = buildForTest({26, 17, 41, 14, 21, 47},
                                {false, true, false, false, false, true});
    cout << "The valid test tree:\n";
    printTree(valid.root, "", true);

    cout << "\nValidators (Problems 8.3, 8.11 to 8.14)\n";
    check("8.3  valid RB tree is a BST", isBST(valid));
    check("8.3  empty tree is a BST", isBST(RBTree()));
    RBTree bad = buildForTest({10, 5, 15}, {false, true, true});
    bad.root->left->key = 20;                       // left child now exceeds the root
    check("8.3  broken BST detected", !isBST(bad));
    check("8.11 four black nodes", countBlackNodes(valid) == 4);
    check("8.11 empty tree has none", countBlackNodes(RBTree()) == 0);
    check("8.12 no red-red in the valid tree", validateProperty4(valid));
    RBTree redRed = buildForTest({10, 5, 15}, {false, true, true});
    redRed.root->left->left = newNode(3);           // RED 3 under RED 5
    redRed.root->left->left->parent = redRed.root->left;
    check("8.12 red-red violation detected", !validateProperty4(redRed));
    check("8.13 uniform black-height", validateBlackHeight(valid));
    check("8.14 valid tree is a valid RB tree", isValidRBTree(valid));
    check("8.14 a red root is rejected", !isValidRBTree(buildForTest({10}, {true})));
    cout << "  black-height of the root = " << computeBlackHeight(valid.root) << "\n";
```

Listing 8.9 (Continued) *Problems 8.11 to 8.15 in one program: the red-black validators, the imperative rotations with parent pointers, a set of checks, and a demonstration of the int-bound pitfall of §8.6.7.*

```
    cout << "\nThe int-bounds pitfall, on a single BLACK node holding INT_MIN\n";
    RBTree edge = buildForTest({INT_MIN}, {false});
    cout << "  isBST with int bounds          : " << (isBST(edge) ? "true" : "false") <<
"\n";
    cout << "  isBSTRepaired, wider bounds    : " << (isBSTRepaired(edge) ? "true" :
"false") << "\n";

    cout << "\nProblem 8.15: leftRotate on the chain 10 -> 20 -> 30\n";
    RBTree t1; for (int k : {10, 20, 30}) t1.bstInsert(k);
    t1.root->red = false;
    printTree(t1.root, "", true);
    leftRotate(t1, t1.root);
    cout << "  after leftRotate(10):\n";
    printTree(t1.root, "", true);
    check("root is now 20", t1.root->key == 20);
    check("20's children are 10 and 30",
          t1.root->left->key == 10 && t1.root->right->key == 30);
    check("10's parent pointer was fixed", t1.root->left->parent == t1.root);

    cout << "\nProblem 8.15: rightRotate on the chain 30 <- 20 <- 10\n";
    RBTree t2; for (int k : {30, 20, 10}) t2.bstInsert(k);
    t2.root->red = false;
    rightRotate(t2, t2.root);
    printTree(t2.root, "", true);
    check("root is now 20", t2.root->key == 20);
    check("20's children are 10 and 30",
          t2.root->left->key == 10 && t2.root->right->key == 30);
    check("30's parent pointer was fixed", t2.root->right->parent == t2.root);

    cout << "\nProblem 8.15: a non-root rotation, leftRotate(30)\n";
    RBTree t3 = buildForTest({20, 10, 30, 25, 35, 33, 40},
                             {false, false, true, false, false, true, true});
    leftRotate(t3, findNode(t3.root, 30));
    printTree(t3.root, "", true);
    check("root is still 20", t3.root->key == 20);
    check("20's right is now 35", t3.root->right->key == 35);
    check("35's left is 30", t3.root->right->left->key == 30);
    check("30's right is 33", t3.root->right->left->right->key == 33);

    cout << "\n=== " << passed << " passed, " << failed << " failed ===\n";
    return 0;
}
```

It prints:

```
The valid test tree:
|-- 26:B
|   |-- 17:R
|   |   |-- 14:B
|   |   `-- 21:B
|   `-- 41:B
|       `-- 47:R

Validators (Problems 8.3, 8.11 to 8.14)
  PASS: 8.3  valid RB tree is a BST
  PASS: 8.3  empty tree is a BST
  PASS: 8.3  broken BST detected
  PASS: 8.11 four black nodes
  PASS: 8.11 empty tree has none
  PASS: 8.12 no red-red in the valid tree
  PASS: 8.12 red-red violation detected
  PASS: 8.13 uniform black-height
  PASS: 8.14 valid tree is a valid RB tree
  PASS: 8.14 a red root is rejected
  black-height of the root = 2

The int-bounds pitfall, on a single BLACK node holding INT_MIN
  isBST with int bounds          : false
  isBSTRepaired, wider bounds    : true

Problem 8.15: leftRotate on the chain 10 -> 20 -> 30
|-- 10:B
|   `-- 20:R
|       `-- 30:R
  after leftRotate(10):
|-- 20:R
|   |-- 10:B
|   `-- 30:R
  PASS: root is now 20
  PASS: 20's children are 10 and 30
  PASS: 10's parent pointer was fixed

Problem 8.15: rightRotate on the chain 30 <- 20 <- 10
|-- 20:R
|   |-- 10:R
|   `-- 30:B
  PASS: root is now 20
  PASS: 20's children are 10 and 30
  PASS: 30's parent pointer was fixed

Problem 8.15: a non-root rotation, leftRotate(30)
|-- 20:B
|   |-- 10:B
|   `-- 35:B
|       |-- 30:R
|       |   |-- 25:B
|       |   `-- 33:R
|       `-- 40:R
  PASS: root is still 20
  PASS: 20's right is now 35
  PASS: 35's left is 30
  PASS: 30's right is 33

=== 20 passed, 0 failed ===
```

Verify the black-height by hand against the printed tree. Down the left: `26:B` is black, `17:R` is red, `14:B` is black, then `NIL` — two black nodes, counting 26 and 14 but not the red 17 and not the `NIL` itself, so black-height 2. Down the right: `26:B`, `41:B`, `47:R`, `NIL` — again two black nodes, 26 and 41. Every path gives 2, which is what the program prints. Four black nodes in the tree altogether — 26, 41, 14, 21 — which is what Problem 8.11 reports.

Then note the non-root rotation: before `leftRotate(30)` the right subtree is $30\{25,35\{33,40\}\}$; afterward it is $35\{30\{25,33\},40\}$. The root 20 is untouched, 33 has moved from being 35's left child to being 30's right child, and the inorder reading 10,20,25,30,33,35,40 is unchanged — which is **Figure 8.7**'s theorem, verified on a concrete tree.

8.15 Problem 8.16 — Matrix-Chain Multiplication, the Table Filled by Hand

8.15.1 The problem

Five matrices are to be multiplied:

$$A_1 : 4 \times 10, \quad A_2 : 10 \times 3, \quad A_3 : 3 \times 12, \quad A_4 : 12 \times 20, \quad A_5 : 20 \times 7.$$

1. Write the dimension sequence d and fill the cost table M and the split table c of `MATRIX-CHAIN-ORDER` by hand, cell by cell, in the order the algorithm fills them.
2. Read the optimal bracketing out of c.
3. Confirm the optimal cost by multiplying out the bracketing, and compare it with the naive left-to-right order.
4. Count the calls made by the plain recursive solution and by the memoized one, and derive both counts in closed form.

8.15.2 What you need from Chapter 7

Multiplying a $p \times q$ matrix by a $q \times r$ matrix takes $p\,q\,r$ scalar multiplications and produces a $p \times r$ matrix. For a chain, the dimensions are listed once as $d = \langle d_0, d_1, \ldots, d_n \rangle$, with A_i of size $d_{i-1} \times d_i$. Let $M[i,j]$ be the least cost of computing $A_i A_{i+1} \cdots A_j$. The last multiplication splits the chain at some k with $i \le k < j$ into $A_i \cdots A_k$ (a $d_{i-1} \times d_k$ matrix) and $A_{k+1} \cdots A_j$ (a $d_k \times d_j$ matrix), and optimal substructure gives

$$M[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{M[i,k] + M[k+1,j] + d_{i-1}\, d_k\, d_j\} & \text{if } i < j. \end{cases} \tag{8.18}$$

The split table $c[i,j]$ records the k that achieved the minimum. **Pseudocode 8.5** fills both tables in order of increasing chain length l, so that every entry a cell needs has already been computed.

Pseudocode 8.5 `MATRIX-CHAIN-ORDER`: every subchain is solved once, shortest first, and the winning split is recorded.

```
MATRIX-CHAIN-ORDER(d, n)
for i = 1 to n
    M[i, i] = 0
for l = 2 to n                        // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        M[i, j] = ∞
        for k = i to j − 1
            q = M[i, k] + M[k + 1, j] + d[i − 1] · d[k] · d[j]
            if q < M[i, j]
                M[i, j] = q
                c[i, j] = k
return M and c
```

8.15.3 Step 1: the dimensions and the diagonal

Reading the sizes off in order: A_1 is 4×10, so $d_0 = 4$ and $d_1 = 10$; A_2 is 10×3, so $d_2 = 3$; then $d_3 = 12$, $d_4 = 20$, $d_5 = 7$. Hence

$$d = \langle 4,10,3,12,20,7 \rangle, \qquad n = 5.$$

Lines 1–2 set $M[1,1] = M[2,2] = \cdots = M[5,5] = 0$: a "chain" of one matrix needs no multiplication.

8.15.4 Step 2: chains of length 2

A chain of two matrices has only one split, so each cell is a single product $d_{i-1}d_id_{i+1}$.

- $M[1,2]$, $k = 1$: $0 + 0 + d_0d_1d_2 = 4 \cdot 10 \cdot 3 = 120$. So $M[1,2] = 120$, $c[1,2] = 1$.
- $M[2,3]$, $k = 2$: $d_1d_2d_3 = 10 \cdot 3 \cdot 12 = 360$. $M[2,3] = 360$, $c[2,3] = 2$.
- $M[3,4]$, $k = 3$: $d_2d_3d_4 = 3 \cdot 12 \cdot 20 = 720$. $M[3,4] = 720$, $c[3,4] = 3$.
- $M[4,5]$, $k = 4$: $d_3d_4d_5 = 12 \cdot 20 \cdot 7 = 1\,680$. $M[4,5] = 1\,680$, $c[4,5] = 4$.

8.15.5 Step 3: chains of length 3

Each cell now has two candidate splits. Every term on the right-hand side is a cell of length 1 or 2, already filled.

$M[1,3]$, the chain $A_1A_2A_3$:

- $k = 1$, meaning $A_1(A_2A_3)$: $M[1,1] + M[2,3] + d_0d_1d_3 = 0 + 360 + 4 \cdot 10 \cdot 12 = 360 + 480 = 840$.
- $k = 2$, meaning $(A_1A_2)A_3$: $M[1,2] + M[3,3] + d_0d_2d_3 = 120 + 0 + 4 \cdot 3 \cdot 12 = 120 + 144 = 264$.

The smaller is 264: $M[1,3] = 264$, $c[1,3] = 2$. Multiplying A_1A_2 first shrinks the 10 to a 3, and every later product is cheaper for it.

$M[2,4]$, the chain $A_2A_3A_4$:

- $k = 2$: $0 + M[3,4] + d_1d_2d_4 = 720 + 10 \cdot 3 \cdot 20 = 720 + 600 = 1\,320$.
- $k = 3$: $M[2,3] + 0 + d_1d_3d_4 = 360 + 10 \cdot 12 \cdot 20 = 360 + 2\,400 = 2\,760$.

$M[2,4] = 1\,320$, $c[2,4] = 2$.

$M[3,5]$, the chain $A_3A_4A_5$:

- $k = 3$: $0 + M[4,5] + d_2d_3d_5 = 1\,680 + 3 \cdot 12 \cdot 7 = 1\,680 + 252 = 1\,932$.
- $k = 4$: $M[3,4] + 0 + d_2d_4d_5 = 720 + 3 \cdot 20 \cdot 7 = 720 + 420 = 1\,140$.

$M[3,5] = 1\,140$, $c[3,5] = 4$.

8.15.6 Step 4: chains of length 4

Three candidates each.

$M[1,4]$, the chain $A_1A_2A_3A_4$:

- $k = 1$: $0 + M[2,4] + d_0d_1d_4 = 1\,320 + 4 \cdot 10 \cdot 20 = 1\,320 + 800 = 2\,120$.
- $k = 2$: $M[1,2] + M[3,4] + d_0d_2d_4 = 120 + 720 + 4 \cdot 3 \cdot 20 = 840 + 240 = 1\,080$.
- $k = 3$: $M[1,3] + 0 + d_0d_3d_4 = 264 + 4 \cdot 12 \cdot 20 = 264 + 960 = 1\,224$.

$M[1,4] = 1\,080$, $c[1,4] = 2$.

$M[2,5]$, the chain $A_2A_3A_4A_5$:

- $k = 2$: $0 + M[3,5] + d_1d_2d_5 = 1\,140 + 10 \cdot 3 \cdot 7 = 1\,140 + 210 = 1\,350$.
- $k = 3$: $M[2,3] + M[4,5] + d_1d_3d_5 = 360 + 1\,680 + 10 \cdot 12 \cdot 7 = 2\,040 + 840 = 2\,880$.
- $k = 4$: $M[2,4] + 0 + d_1d_4d_5 = 1\,320 + 10 \cdot 20 \cdot 7 = 1\,320 + 1\,400 = 2\,720$.

$M[2,5] = 1\,350$, $c[2,5] = 2$.

8.15.7 Step 5: the whole chain

$M[1,5]$, four candidates:

- $k = 1$: $0 + M[2,5] + d_0d_1d_5 = 1\,350 + 4 \cdot 10 \cdot 7 = 1\,350 + 280 = 1\,630$.
- $k = 2$: $M[1,2] + M[3,5] + d_0d_2d_5 = 120 + 1\,140 + 4 \cdot 3 \cdot 7 = 1\,260 + 84 = \mathbf{1\,344}$.
- $k = 3$: $M[1,3] + M[4,5] + d_0d_3d_5 = 264 + 1\,680 + 4 \cdot 12 \cdot 7 = 1\,944 + 336 = 2\,280$.
- $k = 4$: $M[1,4] + 0 + d_0d_4d_5 = 1\,080 + 4 \cdot 20 \cdot 7 = 1\,080 + 560 = 1\,640$.

$M[1,5] = 1\,344$, $c[1,5] = 2$. **Table 8.10** and **Table 8.11** show the finished tables.

Table 8.10 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, $M[1,5] = 1\,344$.

$M[i,j]$	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 8.11 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

8.15.8 Step 6: reading the bracketing, and checking it

`PRINT-OPTIMAL-PARENS` of Chapter 7 reads c from the top:

- $c[1,5] = 2$: the last multiplication is $(A_1A_2) \cdot (A_3A_4A_5)$.
- The left part A_1A_2 has only one bracketing, (A_1A_2).
- The right part: $c[3,5] = 4$, so it is $(A_3A_4) \cdot A_5$.

The optimal order is therefore $((A_1A_2)((A_3A_4)A_5))$. Multiply it out to confirm the cost, tracking the size of each intermediate result:

1. A_1A_2: $(4 \times 10)(10 \times 3)$ costs $4 \cdot 10 \cdot 3 = 120$ and gives a 4×3 matrix.
2. A_3A_4: $(3 \times 12)(12 \times 20)$ costs $3 \cdot 12 \cdot 20 = 720$ and gives 3×20.

3. $(A_3A_4)A_5$: $(3 \times 20)(20 \times 7)$ costs $3 \cdot 20 \cdot 7 = 420$ and gives 3×7.
4. The final product: $(4 \times 3)(3 \times 7)$ costs $4 \cdot 3 \cdot 7 = 84$ and gives 4×7.

Total: $120 + 720 + 420 + 84 = 1\,344$ ✓, exactly $M[1,5]$.

Compare the order a program would use without thinking, left to right, $(((A_1A_2)A_3)A_4)A_5$: 120 for A_1A_2, then $4 \cdot 3 \cdot 12 = 144$, then $4 \cdot 12 \cdot 20 = 960$, then $4 \cdot 20 \cdot 7 = 560$, a total of 1 784 — a third more work for exactly the same product. The optimal order wins by keeping the small dimension 3 on the inside for as long as possible.

8.15.9 Step 7: plain recursion against memoization

Translate Equation (8.18) directly into a recursive function and it recomputes the same subchains over and over. Let $R(n)$ be the number of calls it makes on a chain of n matrices. A chain of one matrix makes one call; a longer chain makes its own call plus, for each of the $n - 1$ splits, a call on a left part of length k and a right part of length $n - k$:

$$R(1) = 1, \qquad R(n) = 1 + \sum_{k=1}^{n-1} (R(k) + R(n-k)) = 1 + 2 \sum_{k=1}^{n-1} R\,(k).$$

Subtract the same equation for $n - 1$: $R(n) - R(n-1) = 2R(n-1)$, so $R(n) = 3R(n-1)$ for $n \geq 3$; and directly $R(2) = 1 + 2 \cdot 1 = 3$. Hence $R(n) = 3^{n-1}$: for our $n = 5$, $3^4 = 81$ calls, and for $n = 17$, more than forty-three million.

The memoized version of Chapter 7 stores each $M[i,j]$ the first time it is computed. Each pair $i < j$ is then *computed* once, and its computation makes $2(j - i)$ calls, two per split; add 1 for the top-level call. Grouping the pairs by their difference $\delta = j - i$, there are $n - \delta$ pairs with that difference, so the total is

$$1 + \sum_{\delta=1}^{n-1} 2\,\delta(n - \delta) = 1 + \frac{n^3 - n}{3}. \tag{8.19}$$

For $n = 5$: $2(1 \cdot 4 + 2 \cdot 3 + 3 \cdot 2 + 4 \cdot 1) = 2 \cdot 20 = 40$, plus 1, is 41 calls. Exponential against cubic — the whole of Chapter 7 in two numbers.

8.15.10 The program

Listing 8.10 implements **Pseudocode 8.5** with a trace of every candidate, prints both tables and the bracketing, counts the calls of the two recursive solutions, and solves the check-yourself chain of §8.15.11.

Listing 8.10 Problem 8.16 in full: `MATRIX-CHAIN-ORDER` with a trace of every candidate split, `PRINT-OPTIMAL-PARENS`, and a count of the calls made by plain and memoized recursion.

```
// mcmtable.cpp -- Problem 8.16: MATRIX-CHAIN-ORDER filled cell by cell, the optimal
// bracketing read from the split table, and memoization against plain recursion.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <climits>
using namespace std;

vector<long long> d;                  // dimensions: A_i is d[i-1] x d[i]
vector<vector<long long>> M;          // M[i][j] = least cost of A_i..A_j
vector<vector<int>> c;                // c[i][j] = the split k that achieved it
long long calls = 0;                  // counts every call of a recursive solver

void matrixChainOrder(int n, bool trace) {
    M.assign(n + 1, vector<long long>(n + 1, 0));
    c.assign(n + 1, vector<int>(n + 1, 0));
    for (int l = 2; l <= n; ++l) {                // chain length, shortest first
        if (trace) cout << "length " << l << "\n";
        for (int i = 1; i <= n - l + 1; ++i) {
            int j = i + l - 1;
            M[i][j] = LLONG_MAX;
            for (int k = i; k <= j - 1; ++k) {
                long long q = M[i][k] + M[k + 1][j] + d[i - 1] * d[k] * d[j];
                if (trace)
                    cout << "  M[" << i << "," << j << "] k=" << k << ": " << M[i][k] << "
+ "
                         << M[k + 1][j] << " + " << d[i - 1] << "*" << d[k] << "*" << d[j]
                         << " = " << q << (q < M[i][j] ? "  <- best so far" : "") << "\n";
                if (q < M[i][j]) { M[i][j] = q; c[i][j] = k; }
            }
        }
    }
}

string parens(int i, int j) {                     // PRINT-OPTIMAL-PARENS
    if (i == j) return "A" + to_string(i);
    return "(" + parens(i, c[i][j]) + parens(c[i][j] + 1, j) + ")";
}
```

Listing 8.10 (Continued) *Problem 8.16 in full: MATRIX-CHAIN-ORDER with a trace of every candidate split, PRINT-OPTIMAL-PARENS, and a count of the calls made by plain and memoized recursion.*

```
// Plain recursion: tries every split and forgets every answer.
long long naive(int i, int j) {
    ++calls;
    if (i == j) return 0;
    long long best = LLONG_MAX;
    for (int k = i; k < j; ++k)
        best = min(best, naive(i, k) + naive(k + 1, j) + d[i - 1] * d[k] * d[j]);
    return best;
}

// Memoized recursion: the same code, but each answer is stored the first time.
vector<vector<long long>> memo;
long long lookup(int i, int j) {
    ++calls;
    if (memo[i][j] >= 0) return memo[i][j];        // already known: no work
    long long best = 0;
    if (i < j) {
        best = LLONG_MAX;
        for (int k = i; k < j; ++k)
            best = min(best, lookup(i, k) + lookup(k + 1, j) + d[i - 1] * d[k] * d[j]);
    }
    return memo[i][j] = best;
}

void printTable(int n, bool costs) {
    cout << (costs ? "  M   " : "  c   ");
    for (int j = 1; j <= n; ++j) cout << setw(7) << j;
    cout << "\n";
    for (int i = 1; i <= n; ++i) {
        cout << "  " << setw(2) << i << "  ";
        for (int j = 1; j <= n; ++j) {
            if (j < i || (!costs && j == i)) cout << setw(7) << "-";
            else cout << setw(7) << (costs ? M[i][j] : (long long)c[i][j]);
        }
        cout << "\n";
    }
}
```

Listing 8.10 (Continued) *Problem 8.16 in full: MATRIX-CHAIN-ORDER with a trace of every candidate split, PRINT-OPTIMAL-PARENS, and a count of the calls made by plain and memoized recursion.*

```
void solve(const vector<long long>& dims, bool trace) {
    d = dims;
    int n = (int)d.size() - 1;
    cout << "d = <";
    for (size_t k = 0; k < d.size(); ++k) cout << (k ? ", " : "") << d[k];
    cout << ">, n = " << n << "\n";
    matrixChainOrder(n, trace);
    printTable(n, true);
    printTable(n, false);
    cout << "  least cost M[1," << n << "] = " << M[1][n]
         << ", bracketing " << parens(1, n) << "\n";
}

int main() {
    solve({4, 10, 3, 12, 20, 7}, true);

    cout << "\nMemoization against plain recursion on the same chain\n";
    int n = 5;
    calls = 0; long long a = naive(1, n);
    cout << "  plain recursion : cost " << a << " after " << calls << " calls\n";
    memo.assign(n + 1, vector<long long>(n + 1, -1));
    calls = 0; long long b = lookup(1, n);
    cout << "  memoized        : cost " << b << " after " << calls << " calls\n";

    cout << "\nGrowth of the call counts (dimensions do not matter for the count)\n";
    cout << "   n   plain recursion   memoized\n";
    for (int m : {5, 8, 11, 14, 17}) {
        d.assign(m + 1, 2);
        calls = 0; naive(1, m); long long p = calls;
        memo.assign(m + 1, vector<long long>(m + 1, -1));
        calls = 0; lookup(1, m); long long q = calls;
        cout << setw(4) << m << setw(18) << p << setw(11) << q << "\n";
    }

    cout << "\nCheck yourself\n";
    solve({2, 3, 6, 4, 5}, false);
    return 0;
}
```

It prints:

```
d = <4, 10, 3, 12, 20, 7>, n = 5
length 2
  M[1,2] k=1: 0 + 0 + 4*10*3 = 120  <- best so far
  M[2,3] k=2: 0 + 0 + 10*3*12 = 360  <- best so far
  M[3,4] k=3: 0 + 0 + 3*12*20 = 720  <- best so far
  M[4,5] k=4: 0 + 0 + 12*20*7 = 1680  <- best so far
length 3
  M[1,3] k=1: 0 + 360 + 4*10*12 = 840  <- best so far
  M[1,3] k=2: 120 + 0 + 4*3*12 = 264  <- best so far
  M[2,4] k=2: 0 + 720 + 10*3*20 = 1320  <- best so far
  M[2,4] k=3: 360 + 0 + 10*12*20 = 2760
  M[3,5] k=3: 0 + 1680 + 3*12*7 = 1932  <- best so far
  M[3,5] k=4: 720 + 0 + 3*20*7 = 1140  <- best so far
length 4
  M[1,4] k=1: 0 + 1320 + 4*10*20 = 2120  <- best so far
  M[1,4] k=2: 120 + 720 + 4*3*20 = 1080  <- best so far
  M[1,4] k=3: 264 + 0 + 4*12*20 = 1224
  M[2,5] k=2: 0 + 1140 + 10*3*7 = 1350  <- best so far
  M[2,5] k=3: 360 + 1680 + 10*12*7 = 2880
  M[2,5] k=4: 1320 + 0 + 10*20*7 = 2720
length 5
  M[1,5] k=1: 0 + 1350 + 4*10*7 = 1630  <- best so far
  M[1,5] k=2: 120 + 1140 + 4*3*7 = 1344  <- best so far
  M[1,5] k=3: 264 + 1680 + 4*12*7 = 2280
  M[1,5] k=4: 1080 + 0 + 4*20*7 = 1640
  M         1      2      3      4      5
   1        0    120    264   1080   1344
   2        -      0    360   1320   1350
   3        -      -      0    720   1140
   4        -      -      -      0   1680
   5        -      -      -      -      0
  c         1      2      3      4      5
   1        -      1      2      2      2
   2        -      -      2      2      2
   3        -      -      -      3      4
   4        -      -      -      -      4
   5        -      -      -      -      -
  least cost M[1,5] = 1344, bracketing ((A1A2)((A3A4)A5))

Memoization against plain recursion on the same chain
  plain recursion : cost 1344 after 81 calls
  memoized        : cost 1344 after 41 calls

Growth of the call counts (dimensions do not matter for the count)
   n   plain recursion   memoized
   5                81         41
   8              2187        169
  11             59049        441
  14           1594323        911
  17          43046721       1633


Check yourself
d = <2, 3, 6, 4, 5>, n = 4
  M         1      2      3      4
   1        0     36     84    124
   2        -      0     72    132
   3        -      -      0    120
   4        -      -      -      0
  c         1      2      3      4
   1        -      1      2      3
   2        -      -      2      3
   3        -      -      -      3
   4        -      -      -      -
  least cost M[1,4] = 124, bracketing (((A1A2)A3)A4)
```

Every "best so far" line is one of the candidates of §8.15.4 to §8.15.7, in the same order, and the two printed tables are **Table 8.10** and **Table 8.11**. The call counts are 3^{n-1} and Equation (8.19) at every size: 81 and 41 for $n = 5$, and 43 046 721 against 1 633 for $n = 17$.

8.15.11 Check yourself

Solve the chain $d = \langle 2,3,6,4,5 \rangle$: four matrices, of sizes 2×3, 3×6, 6×4 and 4×5.

> **ANSWER.** Length 2: $M[1,2] = 2 \cdot 3 \cdot 6 = 36$, $M[2,3] = 3 \cdot 6 \cdot 4 = 72$, $M[3,4] = 6 \cdot 4 \cdot 5 = 120$. Length 3: $M[1,3] = \min(0 + 72 + 2 \cdot 3 \cdot 4,\ 36 + 0 + 2 \cdot 6 \cdot 4) = \min(96,84) = 84$ with $c[1,3] = 2$; $M[2,4] = \min(0 + 120 + 3 \cdot 6 \cdot 5,\ 72 + 0 + 3 \cdot 4 \cdot 5) = \min(210,132) = 132$ with $c[2,4] = 3$. Length 4: $M[1,4] = \min(0 + 132 + 2 \cdot 3 \cdot 5,\ 36 + 120 + 2 \cdot 6 \cdot 5,\ 84 + 0 + 2 \cdot 4 \cdot 5) = \min(162,216,124) = 124$ with $c[1,4] = 3$. **Least cost** 124**, bracketing** $(((A_1A_2)A_3)A_4)$ — here left to right happens to be optimal, because the first dimension 2 is the smallest and every product keeps it on the outside. The last block of the output of **Listing 8.10** confirms both tables.

8.15.12 Common wrong turns on this problem

1. **Filling the table row by row.** $M[1,4]$ needs $M[2,4]$, which is in a later row. The outer loop must run over the chain length l.
2. **Letting k reach j.** The range is $i \le k < j$; with $k = j$ the right part is empty.
3. **Using d_i instead of d_{i-1}.** The left factor of $A_i \cdots A_k$ has d_{i-1} rows. This is why d has $n + 1$ entries and starts at index 0.
4. **Keeping only M.** The cost alone does not say how to bracket. Store c, or you must search the table again to rebuild the answer.
5. **Believing memoization changes the answer.** It changes only the amount of work. Both recursive versions in **Listing 8.10** return 1 344.

8.16 Problem 8.17 — Longest Common Subsequence, the Table and the Walk Back

8.16.1 The problem

Let $X = $ SORTED and $Y = $ STORE.

1. Fill the length table c and the arrow table b of `LCS-LENGTH` by hand, row by row, one cell per line. Break ties by moving up, as Chapter 7 does.
2. Walk the arrows back from the bottom-right corner and write down the longest common subsequence it produces.
3. Is that the only longest common subsequence?
4. State the cost of each phase in terms of $m = |X|$ and $n = |Y|$.

8.16.2 What you need from Chapter 7

A **subsequence** of a string is what remains after crossing out some characters and keeping the rest in order; a **common subsequence** of X and Y is a subsequence of both. Let X_i be the prefix $x_1 \cdots x_i$ and $c[i,j]$ the length of a longest common subsequence of X_i and Y_j. Optimal substructure gives

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max\{c[i-1, j],\ c[i, j-1]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases} \tag{8.20}$$

In words: if the last characters match, that character ends a longest common subsequence of the two prefixes, and we add one to the answer for the prefixes without it; if they do not match, at least one of the two last characters is useless, and we take the better of dropping x_i or dropping y_j. **Pseudocode 8.6** fills c row by row and records in b which case applied: ↖ for a match (value from the diagonal), ↑ for "drop x_i" (value from above), ← for "drop y_j" (value from the left). Ties go up.

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)
for i = 0 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i - 1, j - 1] + 1;  b[i, j] = "↖"
        elseif c[i - 1, j] ≥ c[i, j - 1]
            c[i, j] = c[i - 1, j];          b[i, j] = "↑"
        else
            c[i, j] = c[i, j - 1];          b[i, j] = "←"
return c and b
```

`PRINT-LCS`, **Pseudocode 8.7**, starts at the bottom-right corner and follows the arrows backward. Each ↖ contributes its character; because the walk goes backward, the characters are found last-first and must be printed in reverse.

Pseudocode 8.7 `PRINT-LCS`: follow the arrows back; every diagonal arrow is one character of the answer.

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i - 1, j - 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i - 1, j)
else PRINT-LCS(b, X, i, j - 1)
```

8.16.3 Step 1: the table, one cell per line

Number the characters: $X = x_1 x_6 = $ S, O, R, T, E, D and $Y = y_1 y_5 = $ S, T, O, R, E. Row 0 and column 0 are all 0. For each cell we compare x_i with y_j; on a match we take the diagonal plus one, otherwise the larger of "up" $c[i-1, j]$ and "left" $c[i, j-1]$, going up on a tie.

Row 1, $x_1 = $ S.

- $c[1,1]$: S = S, a match. Diagonal $c[0,0] = 0$, so $c[1,1] = 1$, ↖.
- $c[1,2]$: S ≠ T. Up $c[0,2] = 0$, left $c[1,1] = 1$. Left is larger: 1, ←.

- $c[1,3]$: S ≠ O. Up 0, left 1: 1, ←.
- $c[1,4]$: S ≠ R. Up 0, left 1: 1, ←.
- $c[1,5]$: S ≠ E. Up 0, left 1: 1, ←.

Row 2, $x_2 = $ O.

- $c[2,1]$: O ≠ S. Up $c[1,1] = 1$, left $c[2,0] = 0$: 1, ↑.
- $c[2,2]$: O ≠ T. Up 1, left 1 — a tie, so up: 1, ↑.
- $c[2,3]$: O = O, a match. Diagonal $c[1,2] = 1$: $c[2,3] = 2$, ↖.
- $c[2,4]$: O ≠ R. Up $c[1,4] = 1$, left $c[2,3] = 2$: 2, ←.
- $c[2,5]$: O ≠ E. Up 1, left 2: 2, ←.

Row 3, $x_3 = $ R.

- $c[3,1]$: R ≠ S. Up 1, left 0: 1, ↑.
- $c[3,2]$: R ≠ T. Up 1, left 1, tie: 1, ↑.
- $c[3,3]$: R ≠ O. Up $c[2,3] = 2$, left $c[3,2] = 1$: 2, ↑.
- $c[3,4]$: R = R, a match. Diagonal $c[2,3] = 2$: $c[3,4] = 3$, ↖.
- $c[3,5]$: R ≠ E. Up $c[2,5] = 2$, left $c[3,4] = 3$: 3, ←.

Row 4, $x_4 = $ T.

- $c[4,1]$: T ≠ S. Up 1, left 0: 1, ↑.
- $c[4,2]$: T = T, a match. Diagonal $c[3,1] = 1$: $c[4,2] = 2$, ↖.
- $c[4,3]$: T ≠ O. Up $c[3,3] = 2$, left $c[4,2] = 2$, tie: 2, ↑.
- $c[4,4]$: T ≠ R. Up $c[3,4] = 3$, left 2: 3, ↑.
- $c[4,5]$: T ≠ E. Up $c[3,5] = 3$, left 3, tie: 3, ↑.

Row 5, $x_5 = $ E.

- $c[5,1]$: E ≠ S. Up 1, left 0: 1, ↑.
- $c[5,2]$: E ≠ T. Up $c[4,2] = 2$, left 1: 2, ↑.
- $c[5,3]$: E ≠ O. Up 2, left 2, tie: 2, ↑.
- $c[5,4]$: E ≠ R. Up 3, left 2: 3, ↑.
- $c[5,5]$: E = E, a match. Diagonal $c[4,4] = 3$: $c[5,5] = 4$, ↖.

Row 6, $x_6 = $ D. D occurs nowhere in Y, so no cell of this row matches.

- $c[6,1]$: up 1, left 0: 1, ↑.
- $c[6,2]$: up 2, left 1: 2, ↑.
- $c[6,3]$: up 2, left 2, tie: 2, ↑.
- $c[6,4]$: up 3, left 2: 3, ↑.
- $c[6,5]$: up $c[5,5] = 4$, left 3: 4, ↑.

The length of a longest common subsequence is $c[6,5] = 4$. **Table 8.12** shows the finished tables, each cell carrying its value and its arrow.

Table 8.12 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 ↑

8.16.4 Step 2: the walk back

Start at $(i, j) = (6,5)$ and follow the arrows until an index reaches 0:

1. (6,5): ↑. Move to (5,5).
2. (5,5): ↖. **Take $x_5 = $ E.** Move to (4,4).
3. (4,4): ↑. Move to (3,4).
4. (3,4): ↖. **Take $x_3 = $ R.** Move to (2,3).
5. (2,3): ↖. **Take $x_2 = $ O.** Move to (1,2).
6. (1,2): ←. Move to (1,1).
7. (1,1): ↖. **Take $x_1 = $ S.** Move to (0,0) and stop.

The characters were taken in the order E, R, O, S; reversed, the answer is **SORE**. Check it by hand: in SORTED, S, O, R, E appear in that order (crossing out T and D); in STORE, S, O, R, E appear in that order (crossing out T). Its length 4 agrees with $c[6,5]$.

8.16.5 Step 3: is SORE the only answer?

The table says only that the longest length is 4. To see that SORE is the only common subsequence of that length, consider where the letter T can sit. In $Y = $ STORE the T comes *before* O and R, while in $X = $ SORTED it comes *after* them. So a common subsequence containing T can contain neither O nor R; the longest such is STE, of length 3. Every common subsequence of length 4 therefore avoids T, and avoids D, which is not in Y — leaving the letters S, O, R, E of X in their only order. So SORE is the unique longest common subsequence. Enumerating every subsequence of both strings by machine confirms this, and also that the length-3 common subsequences are exactly ORE, SOE, SOR, SRE and STE.

When there *are* several longest common subsequences, the one the walk produces depends on the tie rule: going up on a tie, as here, or left, may lead to different — equally long — answers. The algorithm promises *a* longest common subsequence, not a particular one.

8.16.6 Step 4: the cost

Filling the table visits each of the $m \times n$ cells once and does a constant amount of work there, so `LCS-LENGTH` takes $\Theta(mn)$ time and $\Theta(mn)$ space: here $6 \times 5 = 30$ cells. The walk back decreases i, j or both at every step, so it takes at most $m + n$ steps — here 7 steps, against a limit of 11.

Chapter 7 shows how to keep only two rows when the length alone is wanted, cutting the space to $\Theta(\min(m, n))$; the walk back needs the whole b table, which is why the program keeps it.

8.16.7 The program

Listing 8.11 implements **Pseudocode 8.6** with a line of trace per cell, prints the combined table with `D`, `U` and `L` standing for ↖, ↑ and ←, and implements **Pseudocode 8.7** as a loop.

Listing 8.11 Problem 8.17 in full: `LCS-LENGTH` with a trace of every cell, the combined table of values and arrows, and the walk back that produces one longest common subsequence.

```
// lcstable.cpp -- Problem 8.17: LCS-LENGTH filled cell by cell, the table of arrows,
// and the reconstruction of one longest common subsequence by walking back.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

// b[i][j]: 'D' = diagonal (copy), 'U' = up (skip x_i), 'L' = left (skip y_j)
vector<vector<int>>  C;
vector<vector<char>> B;

void lcsLength(const string& X, const string& Y, bool trace) {
    int m = (int)X.size(), n = (int)Y.size();
    C.assign(m + 1, vector<int>(n + 1, 0));        // row 0 and column 0 stay 0
    B.assign(m + 1, vector<char>(n + 1, ' '));
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (X[i - 1] == Y[j - 1]) {                 // strings are 0-based in C++
                C[i][j] = C[i - 1][j - 1] + 1; B[i][j] = 'D';
            } else if (C[i - 1][j] >= C[i][j - 1]) {    // ties go up, as in Chapter 7
                C[i][j] = C[i - 1][j];         B[i][j] = 'U';
            } else {
                C[i][j] = C[i][j - 1];         B[i][j] = 'L';
            }
            if (trace) {
                cout << "  c[" << i << "," << j << "]  x" << i << "=" << X[i - 1]
                     << " y" << j << "=" << Y[j - 1] << "  ";
                if (B[i][j] == 'D')
                    cout << "match:    c[" << i - 1 << "," << j - 1 << "]+1 = " << C[i][j]
<< "  D\n";
                else
                    cout << "no match: max(up " << C[i - 1][j] << ", left " << C[i][j - 1]
                         << ") = " << C[i][j] << "  " << B[i][j] << "\n";
            }
        }
    }
}
```

Listing 8.11 (Continued) *Problem 8.17 in full: LCS-LENGTH with a trace of every cell, the combined table of values and arrows, and the walk back that produces one longest common subsequence.*

```
// PRINT-LCS: follow the arrows back from (i,j); every D contributes a character.
string walkBack(const string& X, int i, int j, bool trace) {
    string out;
    while (i > 0 && j > 0) {
        char a = B[i][j];
        if (trace) cout << "  at (" << i << "," << j << ") arrow " << a;
        if (a == 'D') {
            out = X[i - 1] + out;
            if (trace) cout << "  take " << X[i - 1] << "\n";
            --i; --j;
        } else {
            if (trace) cout << "\n";
            if (a == 'U') --i; else --j;
        }
    }
    return out;
}

void printTable(const string& X, const string& Y) {
    int m = (int)X.size(), n = (int)Y.size();
    cout << "      j:" << setw(4) << 0 << "  ";                        // column numbers,
then the letters of Y
    for (int j = 1; j <= n; ++j) cout << setw(3) << j << "  ";
    cout << "\n              ";
    for (int j = 1; j <= n; ++j) cout << "  " << Y[j - 1] << "  ";
    cout << "\n  i=0    ";
    for (int j = 0; j <= n; ++j) cout << setw(3) << 0 << "  ";
    cout << "\n";
    for (int i = 1; i <= m; ++i) {
        cout << "  i=" << i << " " << X[i - 1] << "  " << setw(3) << 0 << "  ";
        for (int j = 1; j <= n; ++j) cout << setw(3) << C[i][j] << B[i][j] << " ";
        cout << "\n";
    }
}

void solve(const string& X, const string& Y, bool trace) {
    cout << "X = " << X << " (m = " << X.size() << "), Y = " << Y
         << " (n = " << Y.size() << ")\n";
    lcsLength(X, Y, trace);
    printTable(X, Y);
    string z = walkBack(X, (int)X.size(), (int)Y.size(), trace);
    cout << "  LCS length = " << C[X.size()][Y.size()] << ", one LCS = " << z << "\n";
}
```

Listing 8.11 (Continued) *Problem 8.17 in full: LCS-LENGTH with a trace of every cell, the combined table of values and arrows, and the walk back that produces one longest common subsequence.*

```
int main() {
    solve("SORTED", "STORE", true);
    cout << "\nCheck yourself\n";
    solve("HEAPS", "SHAPE", false);
    return 0;
}
```

It prints:

```
X = SORTED (m = 6), Y = STORE (n = 5)
  c[1,1]  x1=S y1=S  match:    c[0,0]+1 = 1  D
  c[1,2]  x1=S y2=T  no match: max(up 0, left 1) = 1  L
  c[1,3]  x1=S y3=O  no match: max(up 0, left 1) = 1  L
  c[1,4]  x1=S y4=R  no match: max(up 0, left 1) = 1  L
  c[1,5]  x1=S y5=E  no match: max(up 0, left 1) = 1  L
  c[2,1]  x2=O y1=S  no match: max(up 1, left 0) = 1  U
  c[2,2]  x2=O y2=T  no match: max(up 1, left 1) = 1  U
  c[2,3]  x2=O y3=O  match:    c[1,2]+1 = 2  D
  c[2,4]  x2=O y4=R  no match: max(up 1, left 2) = 2  L
  c[2,5]  x2=O y5=E  no match: max(up 1, left 2) = 2  L
  c[3,1]  x3=R y1=S  no match: max(up 1, left 0) = 1  U
  c[3,2]  x3=R y2=T  no match: max(up 1, left 1) = 1  U
  c[3,3]  x3=R y3=O  no match: max(up 2, left 1) = 2  U
  c[3,4]  x3=R y4=R  match:    c[2,3]+1 = 3  D
  c[3,5]  x3=R y5=E  no match: max(up 2, left 3) = 3  L
  c[4,1]  x4=T y1=S  no match: max(up 1, left 0) = 1  U
  c[4,2]  x4=T y2=T  match:    c[3,1]+1 = 2  D
  c[4,3]  x4=T y3=O  no match: max(up 2, left 2) = 2  U
  c[4,4]  x4=T y4=R  no match: max(up 3, left 2) = 3  U
  c[4,5]  x4=T y5=E  no match: max(up 3, left 3) = 3  U
  c[5,1]  x5=E y1=S  no match: max(up 1, left 0) = 1  U
  c[5,2]  x5=E y2=T  no match: max(up 2, left 1) = 2  U
  c[5,3]  x5=E y3=O  no match: max(up 2, left 2) = 2  U
  c[5,4]  x5=E y4=R  no match: max(up 3, left 2) = 3  U
  c[5,5]  x5=E y5=E  match:    c[4,4]+1 = 4  D
  c[6,1]  x6=D y1=S  no match: max(up 1, left 0) = 1  U
  c[6,2]  x6=D y2=T  no match: max(up 2, left 1) = 2  U
  c[6,3]  x6=D y3=O  no match: max(up 2, left 2) = 2  U
  c[6,4]  x6=D y4=R  no match: max(up 3, left 2) = 3  U
  c[6,5]  x6=D y5=E  no match: max(up 4, left 3) = 4  U
      j:   0    1    2    3    4    5
                S    T    O    R    E
  i=0      0    0    0    0    0    0
  i=1 S    0    1D   1L   1L   1L   1L
  i=2 O    0    1U   1U   2D   2L   2L
  i=3 R    0    1U   1U   2U   3D   3L
  i=4 T    0    1U   2D   2U   3U   3U
  i=5 E    0    1U   2U   2U   3U   4D
  i=6 D    0    1U   2U   2U   3U   4U
  at (6,5) arrow U
  at (5,5) arrow D  take E
  at (4,4) arrow U
  at (3,4) arrow D  take R
  at (2,3) arrow D  take O
  at (1,2) arrow L
  at (1,1) arrow D  take S
  LCS length = 4, one LCS = SORE

Check yourself
X = HEAPS (m = 5), Y = SHAPE (n = 5)
      j:   0    1    2    3    4    5
                S    H    A    P    E
  i=0      0    0    0    0    0    0
  i=1 H    0    0U   1D   1L   1L   1L
  i=2 E    0    0U   1U   1U   1U   2D
  i=3 A    0    0U   1U   2D   2L   2U
  i=4 P    0    0U   1U   2U   3D   3L
  i=5 S    0    1D   1U   2U   3U   3U
  LCS length = 3, one LCS = HAP
```

The thirty trace lines are the thirty cells of §8.16.3, in the same order and with the same arrows; the table is **Table 8.12**; and the seven "at" lines are the seven steps of §8.16.4.

8.16.8 Check yourself

Find a longest common subsequence of $X = $ HEAPS and $Y = $ SHAPE by filling the table by hand.

> **ANSWER.** The rows, as value and arrow for $j = 1, \ldots, 5$ (S, H, A, P, E): H: 0 ↑, 1 ↖, 1 ←, 1 ←, 1 ←. E: 0 ↑, 1 ↑, 1 ↑, 1 ↑, 2 ↖. A: 0 ↑, 1 ↑, 2 ↖, 2 ←, 2 ↑. P: 0 ↑, 1 ↑, 2 ↑, 3 ↖, 3 ←. S: 1 ↖, 1 ↑, 2 ↑, 3 ↑, 3 ↑. The length is $c[5,5] = 3$, and the walk back from (5,5) goes up to (4,5), left to (4,4), takes P there, takes A at (3,3), moves diagonally to (2,2), goes up to (1,2), and takes H: **HAP**. It is the only common subsequence of length 3: S is first in Y but last in X, and E is last in Y but second in X, so neither can join H, A and P. The second block of the output of **Listing 8.11** confirms the table.

8.16.9 Common wrong turns on this problem

1. **Taking the diagonal on a mismatch.** The diagonal is used only when $x_i = y_j$; on a mismatch the value comes from above or from the left, never from the diagonal plus anything.
2. **Printing the walk-back characters in the order found.** They come out last-first: EROS instead of SORE. Reverse them, or print after the recursive call as **Pseudocode 8.7** does.
3. **Off-by-one between the table and the strings.** The table is indexed from 1 for characters, but C++ strings start at 0, so x_i is `X[i-1]`. **Listing 8.11** marks the line.
4. **Expecting the tie rule to change the length.** It changes which longest subsequence is found, never how long it is.
5. **Confusing a subsequence with a substring.** A substring must be contiguous; SORE is not a substring of SORTED, but it is a subsequence.

8.17 Twenty-Five Quick Checks

Twenty-five questions with one-line answers, covering the central ideas of Chapters 1 to 7. Cover the answers, work through the list, and check yourself. Anything you miss names the section you should reread.

1. **What are the five phases of the course's problem-solving framework?** Understand, abstract, design, analyze, implement. (Ch. 1)
2. **What does the RAM model charge for an array subscript?** One unit — constant time, like any arithmetic or comparison. (Ch. 2)
3. **State the loop invariant of insertion sort.** At the start of each iteration of the outer loop, the subarray $A[1..j-1]$ holds the original elements of $A[1..j-1]$ in sorted order. (Ch. 2)
4. **The three obligations of a loop invariant?** Initialization, maintenance, termination. (Ch. 2)
5. **Best and worst case of insertion sort, with the input that causes each?** $\Theta(n)$ on an already-sorted array; $\Theta(n^2)$ on a reverse-sorted array. (Ch. 2)
6. **How many comparisons does binary search make on n elements?** $\Theta(\lg n)$ — precisely $\lfloor \lg n \rfloor + 1$ in the worst case. (Ch. 2)
7. **Define $f(n) = O(g(n))$.** There exist constants $c > 0$ and n_0 such that $0 \le f(n) \le c\, g(n)$ for all $n \ge n_0$. (Ch. 3)
8. **Difference between O and o?** O allows the bound to be tight; o requires $f(n)/g(n) \to 0$, so the bound is strictly loose. (Ch. 3)
9. **Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?** Yes and no: $2^{n+1} = 2 \cdot 2^n$, but $2^{2n} = (2^n)^2$ grows faster than any constant multiple. (Ch. 3)

10. **Evaluate $\sum_{i=1}^{n} i$ and $\sum_{i=0}^{n} 2^i$.** $\frac{n(n+1)}{2}$ and $2^{n+1} - 1$. (Ch. 3)
11. **The three cases of the Master Theorem, in one line each.** $f(n) = O(n^{\log_b a - \varepsilon}) \Rightarrow \Theta(n^{\log_b a})$; $f(n) = \Theta(n^{\log_b a}) \Rightarrow \Theta(n^{\log_b a}\lg n)$; $f(n) = \Omega(n^{\log_b a + \varepsilon})$ with the regularity condition $\Rightarrow \Theta(f(n))$. (Ch. 3)
12. **Solve** $T(n) = 2T(n/2) + n$**.** $a = b = 2$, $\log_b a = 1$, $f(n) = \Theta(n^1)$: case 2, so $T(n) = \Theta(n\lg n)$. (Ch. 3)
13. **Solve** $T(n) = T(n/2) + \Theta(1)$**.** $\log_b a = 0$, $f(n) = \Theta(n^0)$: case 2, so $T(n) = \Theta(\lg n)$ — binary search. (Ch. 3)
14. **Why is merge sort $\Theta(n\lg n)$ in *every* case?** The split is by position, so both halves always have $n/2$ elements regardless of the data; the recurrence has no best or worst variant. (Ch. 3)
15. **State the max-heap property, and why it is not the BST property.** $A[\text{parent}(i)] \geq A[i]$ — a parent-and-children rule with no left/right distinction, so a purely local check is correct for a heap and wrong for a BST. (Ch. 5, §8.6.6)
16. **Why is `BUILD-MAX-HEAP` $\Theta(n)$ and not $\Theta(n\lg n)$?** Most nodes are near the bottom, where `MAX-HEAPIFY` costs almost nothing; summing $\sum_h \lceil n/2^{h+1} \rceil\, O(h)$ gives $O(n)$. (Ch. 5, §8.3.6)
17. **Heapsort's running time and space?** $\Theta(n\lg n)$ always, and $O(1)$ extra space — it sorts in place. (Ch. 5)
18. **Quicksort's worst case, and the input that causes it with a last-element pivot?** $\Theta(n^2)$, on an already-sorted array: exactly $\frac{n(n-1)}{2}$ comparisons. (Ch. 5, §8.4.5)
19. **What does randomizing the pivot buy, and what does it not?** Expected $\Theta(n\lg n)$ on *every* input; it does not change the $\Theta(n^2)$ worst case, only its probability. (Ch. 5)
20. **The comparison-sort lower bound, and why?** $\Omega(n\lg n)$: a decision tree with $n!$ leaves has height at least $\lg(n!) = \Theta(n\lg n)$. (Ch. 5)
21. **Why is a BST's inorder walk sorted?** Every left-subtree value is below the root and every right-subtree value above it, so left-root-right emits them in increasing order; by induction the whole walk is sorted. (Ch. 6, §8.7.3)
22. **The AVL height bound?** $h \leq 1.4405\lg(n+2) - 1.3277$, from the minimum-nodes recurrence $N(h) = 1 + N(h-1) + N(h-2)$, whose solution is $N(h) = F(h+3) - 1$. (Ch. 6)
23. **How many rotations can one AVL insertion cause? One deletion?** At most two for an insertion; up to $\Theta(\lg n)$ single rotations for a deletion. (Ch. 6, §8.13.4)
24. **What makes a problem suitable for dynamic programming rather than plain divide and conquer?** Overlapping subproblems together with optimal substructure; divide and conquer suits *independent* subproblems. (Ch. 7, Problems 8.16 and 8.17)
25. **Memoization versus bottom-up tabulation — the difference in one line?** Both store each subproblem's answer once; memoization keeps the recursion and fills on demand, tabulation removes the recursion and fills in dependency order. (Ch. 7, §8.15.9)

8.18 Further Worked Problems: an Order-Statistic Tree

The eight problems of this section, Problems 8.18 to 8.25, have the same shape as Problems 8.3 to 8.10 but a different invariant, so that remembering the earlier solutions will not be enough and understanding them will. Try each one before reading its solution.

8.18.1 The problems

The working type is an **order-statistic tree**: a binary search tree in which every node stores, in addition to its integer value, two derived fields — `h`, the height of its own subtree, and `size`, the number of nodes in its own subtree. Both fields are maintained by the single constructor `node(v, l, r)`.

Problem 8.18. Write `isOrderStatisticTree(OSTree t)`, with a recursive helper, which returns `true` exactly when *all three* of the following hold at every node: the binary-search-tree property; the stored `size` equals $1 + \text{size(left)} + \text{size(right)}$; and the stored `h` equals $1 + \max(h(\text{left}), h(\text{right}))$. The helper must make **one pass**. State its complexity.

Problem 8.19. Write `printBetween(OSTree t, int lo, int hi)`, which prints, in **ascending** order, every value v of t with $lo \le v \le hi$. It must **not** build a list of the values, and it must **not** visit a subtree that cannot contain a value in the range.

Problem 8.20. Write `select(OSTree t, int k)`, which returns the k-th smallest value of t, with k counted from 1. It must follow **at most one path** from the root. State why the method is $O(h)$.

Problem 8.21. Write `isWeightBalanced(OSTree t)`, which returns `true` exactly when, at every node, $\text{size(left)} \le 2\,\text{size(right)} + 1$ and $\text{size(right)} \le 2\,\text{size(left)} + 1$. One pass. Then answer in one sentence: *is every AVL tree weight-balanced in this sense?* Justify with an example or a proof.

Problem 8.22. Write `insertOS(int n, OSTree t)`, which inserts n while preserving the **height**-balance (AVL) invariant *and* the correctness of both stored fields. It must use the two functions of Problems 8.23 and 8.24.

Problem 8.23. Write `rebalanceForLeft(OSTree t)`, called when the left subtree of t may have grown taller by one notch.

Problem 8.24. Write `rebalanceForRight(OSTree t)`, the mirror image.

Problem 8.25. Write `deleteOS(OSTree t, int x)`, which deletes x while preserving the AVL invariant and both stored fields.

Table 8.13 pairs each problem with the earlier problem it extends, and says what is new.

Table 8.13 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

8.18.2 How each solution is derived

Problem 8.18. Extend the bounded-interval recursion of §8.6.4 so that it *returns* the true height and size instead of a bare boolean — the sentinel idea of §8.9.4, generalized to a small record. At

each node: check the bounds, recurse left with the upper bound replaced by the node's value, recurse right with the lower bound replaced, then compare the two computed fields against the two stored ones, then check $|h_R - h_L| \leq 1$. One pass, $\Theta(n)$ time, $O(h)$ stack.

Problem 8.19. The ordinary inorder walk, with two pruning tests. Descend left only when the node's value exceeds lo — if the node is already $\leq lo$, every value to its left is smaller still and none can be in range. Descend right only when the node's value is below hi, by the mirror argument. Cost: $O(h + k)$ where k is the number of values reported, which is optimal.

Problem 8.20. The `size` field turns selection into a descent. At a node, let $\ell = \text{size(left)}$. If $k \leq \ell$, the answer is the k-th smallest of the left subtree. If $k = \ell + 1$, the answer is this node's value — there are exactly ℓ values below it. Otherwise the answer is the $(k - \ell - 1)$-th smallest of the right subtree, because we have discarded the ℓ left values *and* the node itself. One recursive call per level, so $O(h)$.

Problem 8.21. The sentinel technique with sizes: return the subtree's size, or -1 to mean "already failed". -1 is free, because a size is never negative. $\Theta(n)$.

And the one-sentence question: **no**, an AVL tree need not be weight-balanced in this sense. The counterexample is constructed rather than found. Take a root whose left subtree is *perfect* of height 2 — seven nodes — and whose right subtree is a *minimal* AVL tree of height 1 — two nodes. The heights differ by exactly 1, so the root is height-balanced and so is everything below it; but $\text{size(left)} = 7$ and $2\,\text{size(right)} + 1 = 2 \cdot 2 + 1 = 5$, and $7 > 5$. Height balance bounds the *depths* of the two sides; it does not bound their *populations*, because a subtree of a given height may hold as few as $N(h)$ nodes or as many as $2^{h+1} - 1$.

Problems 8.22 to 8.25. Structurally identical to §§8.10–8.13. The only new obligation is that `node(v,l,r)` must fix `size` as well as `h`, and it does — which is why the rotations, the rebalancers, `insertOS` and `deleteOS` need not mention `size` even once. That is the payoff of the single-constructor discipline of §8.5: adding a second derived field cost one line.

8.18.3 Complete solutions

Listing 8.12 solves all eight problems in one program. Its `main` builds an eleven-node tree, deliberately corrupts one stored field to show the validator of Problem 8.18 at work, and ends with a stress test of two thousand insertions and deletions.

Listing 8.12 Complete solutions to Problems 8.18 to 8.25, with a deliberate corruption of one stored field to show the validator of Problem 8.18 catching it, and a two-thousand-element stress test.

```
// ostree.cpp -- complete solutions to Problems 8.18 to 8.25 of section 8.18.
// An order-statistic tree: an AVL tree in which every node also stores the
// number of nodes in its own subtree.
#include <iostream>
#include <iomanip>
#include <string>
#include <stdexcept>
#include <vector>
#include <algorithm>
using namespace std;

struct OSNode;
using OSTree = OSNode*;
struct OSNode { int value; OSTree left, right; int h, size; };

OSTree emptyTree()       { return nullptr; }
bool   isEmpty(OSTree t) { return t == nullptr; }
int    height(OSTree t)  { return isEmpty(t) ? -1 : t->h; }
int    sizeOf(OSTree t)  { return isEmpty(t) ?  0 : t->size; }
int    value(OSTree t)   { return t->value; }
OSTree leftOf(OSTree t)  { return t->left; }
OSTree rightOf(OSTree t) { return t->right; }

// The single constructor. It fixes BOTH derived fields, so neither can go stale.
OSTree node(int v, OSTree l, OSTree r) {
    OSTree t = new OSNode;
    t->value = v; t->left = l; t->right = r;
    t->h    = 1 + max(height(l), height(r));
    t->size = 1 + sizeOf(l) + sizeOf(r);
    return t;
}
OSTree leaf(int v) { return node(v, emptyTree(), emptyTree()); }
int balanceFactor(OSTree t) { return height(rightOf(t)) - height(leftOf(t)); }

string shape(OSTree t) {
    if (isEmpty(t)) return ".";
    string s = to_string(value(t)) + "[" + to_string(sizeOf(t)) + "]";
    if (isEmpty(leftOf(t)) && isEmpty(rightOf(t))) return s;
    return s + "{" + shape(leftOf(t)) + "," + shape(rightOf(t)) + "}";
}
```

Listing 8.12 (Continued) *Complete solutions to Problems 8.18 to 8.25, with a deliberate corruption of one stored field to show the validator of Problem 8.18 catching it, and a two-thousand-element stress test.*

```
// ===== Problem 8.18: validate the three invariants in one pass =====
// Returns the subtree's true (height, size), or reports the first fault.
struct Fields { int h, size; };
static bool validate(OSTree t, long long lo, long long hi, Fields& out, string& why) {
    if (isEmpty(t)) { out = {-1, 0}; return true; }
    if (value(t) <= lo) { why = "node " + to_string(value(t)) + " must exceed " +
to_string(lo); return false; }
    if (value(t) >= hi) { why = "node " + to_string(value(t)) + " must be below " +
to_string(hi); return false; }
    Fields L, R;
    if (!validate(leftOf(t),  lo, value(t), L, why)) return false;
    if (!validate(rightOf(t), value(t), hi, R, why)) return false;
    int trueH = 1 + max(L.h, R.h), trueSize = 1 + L.size + R.size;
    if (t->h != trueH) {
        why = "node " + to_string(value(t)) + " stores height " + to_string(t->h)
            + " but the truth is " + to_string(trueH); return false;
    }
    if (t->size != trueSize) {
        why = "node " + to_string(value(t)) + " stores size " + to_string(t->size)
            + " but the truth is " + to_string(trueSize); return false;
    }
    if (abs(R.h - L.h) > 1) {
        why = "node " + to_string(value(t)) + " has balance factor " + to_string(R.h -
L.h);
        return false;
    }
    out = {trueH, trueSize};
    return true;
}
bool isOrderStatisticTree(OSTree t, string& why) {
    Fields f; why = "none";
    return validate(t, -2147483649LL, 2147483648LL, f, why);
}

// ===== Problem 8.19: print every value in [lo,hi], ascending, no list built =====
void printBetween(OSTree t, int lo, int hi) {
    if (isEmpty(t)) return;
    if (value(t) > lo) printBetween(leftOf(t), lo, hi);   // prune: nothing < lo on the
left
    if (lo <= value(t) && value(t) <= hi) cout << value(t) << ' ';
    if (value(t) < hi) printBetween(rightOf(t), lo, hi);  // prune: nothing > hi on the
right
}
```

Listing 8.12 (Continued) *Complete solutions to Problems 8.18 to 8.25, with a deliberate corruption of one stored field to show the validator of Problem 8.18 catching it, and a two-thousand-element stress test.*

```
// ===== Problem 8.20: the k-th smallest value, one downward path, O(h) =====
long long probes = 0;
int select_(OSTree t, int k) {                  // k is 1-based
    if (isEmpty(t)) throw invalid_argument("k is out of range");
    ++probes;
    int leftCount = sizeOf(leftOf(t));
    if (k <= leftCount)      return select_(leftOf(t), k);
    if (k == leftCount + 1)  return value(t);
    return select_(rightOf(t), k - leftCount - 1);   // discard the left side AND t
}

// ===== Problem 8.21: the weight-balance predicate, one pass =====
// Invariant: at every node, size(left) <= 2*size(right)+1 and vice versa.
const int WB_FAIL = -1;
static int checkedSize(OSTree t, string& why) {
    if (isEmpty(t)) return 0;
    int sl = checkedSize(leftOf(t),  why); if (sl == WB_FAIL) return WB_FAIL;
    int sr = checkedSize(rightOf(t), why); if (sr == WB_FAIL) return WB_FAIL;
    if (sl > 2 * sr + 1 || sr > 2 * sl + 1) {
        why = "node " + to_string(value(t)) + ": left size " + to_string(sl)
            + ", right size " + to_string(sr);
        return WB_FAIL;
    }
    return 1 + sl + sr;
}
bool isWeightBalanced(OSTree t, string& why) { why = "none"; return checkedSize(t, why) !=
WB_FAIL; }

// ===== Problems 8.23 and 8.24: the two rebalancers =====
long long rotations = 0;
OSTree rotateRight(OSTree t) {
    ++rotations; OSTree l = leftOf(t);
    return node(value(l), leftOf(l), node(value(t), rightOf(l), rightOf(t)));
}
OSTree rotateLeft(OSTree t) {
    ++rotations; OSTree r = rightOf(t);
    return node(value(r), node(value(t), leftOf(t), leftOf(r)), rightOf(r));
}
OSTree rebalanceForLeft(OSTree t) {
    if (balanceFactor(t) >= -1) return t;
    OSTree l = leftOf(t);
    if (balanceFactor(l) <= 0) return rotateRight(t);
    return rotateRight(node(value(t), rotateLeft(l), rightOf(t)));
}
OSTree rebalanceForRight(OSTree t) {
    if (balanceFactor(t) <= 1) return t;
    OSTree r = rightOf(t);
```

Listing 8.12 (Continued) *Complete solutions to Problems 8.18 to 8.25, with a deliberate corruption of one stored field to show the validator of Problem 8.18 catching it, and a two-thousand-element stress test.*

```
    if (balanceFactor(r) >= 0) return rotateLeft(t);
    return rotateLeft(node(value(t), leftOf(t), rotateRight(r)));
}

// ===== Problem 8.22: insert, keeping height balance and the size field =====
OSTree insertOS(int n, OSTree t) {
    if (isEmpty(t)) return leaf(n);
    if (n < value(t)) return rebalanceForLeft (node(value(t), insertOS(n, leftOf(t)),
rightOf(t)));
    if (n > value(t)) return rebalanceForRight(node(value(t), leftOf(t), insertOS(n,
rightOf(t))));
    return t;
}

// ===== Problem 8.25: delete, keeping height balance and the size field =====
int maxOf(OSTree t) { return isEmpty(rightOf(t)) ? value(t) : maxOf(rightOf(t)); }
OSTree deleteOS(OSTree t, int x) {
    if (isEmpty(t)) return t;
    if (x < value(t)) return rebalanceForRight(node(value(t), deleteOS(leftOf(t), x),
rightOf(t)));
    if (x > value(t)) return rebalanceForLeft (node(value(t), leftOf(t),
deleteOS(rightOf(t), x)));
    if (isEmpty(leftOf(t)))  return rightOf(t);
    if (isEmpty(rightOf(t))) return leftOf(t);
    int m = maxOf(leftOf(t));
    return rebalanceForRight(node(m, deleteOS(leftOf(t), m), rightOf(t)));
}

int main() {
    OSTree t = emptyTree();
    for (int v : {50, 30, 70, 20, 40, 60, 80, 10, 35, 45, 65}) t = insertOS(v, t);
    string why;
    cout << "8.18 shape            : " << shape(t) << "\n";
    cout << "8.18 valid OS tree    : " << (isOrderStatisticTree(t, why) ? "yes" : "no")
         << "   (fault: " << why << ")\n";
    t->size = 99;                                   // corrupt one field on purpose
    cout << "8.18 after corrupting : " << (isOrderStatisticTree(t, why) ? "yes" : "no")
         << "   (fault: " << why << ")\n";
    t->size = 11;                                   // put it back
    cout << "8.18 repaired         : " << (isOrderStatisticTree(t, why) ? "yes" : "no") <<
"\n\n";

    cout << "8.19 all values       : "; printBetween(t, -1000, 1000); cout << "\n";
    cout << "8.19 in [30,60]       : "; printBetween(t,    30,   60); cout << "\n";
    cout << "8.19 in [41,44]       : "; printBetween(t,    41,   44); cout <<
"(nothing)\n\n";
```

Listing 8.12 (Continued) *Complete solutions to Problems 8.18 to 8.25, with a deliberate corruption of one stored field to show the validator of Problem 8.18 catching it, and a two-thousand-element stress test.*

```
    cout << "8.20 k-th smallest    :";
    for (int k = 1; k <= sizeOf(t); ++k) cout << ' ' << select_(t, k);
    cout << "\n";
    probes = 0; int v6 = select_(t, 6);
    cout << "8.20 select(6) = " << v6 << " after " << probes
         << " node probes (n = " << sizeOf(t) << ", h = " << height(t) << ")\n\n";

    cout << "8.21 weight-balanced  : " << (isWeightBalanced(t, why) ? "yes" : "no")
         << "   (fault: " << why << ")\n";
    // An AVL tree that is NOT weight-balanced: left perfect of height 2 (7 nodes),
    // right a minimal AVL tree of height 1 (2 nodes).
    OSTree skew = node(100,
                        node(40, node(20, leaf(10), leaf(30)), node(60, leaf(50),
leaf(70))),
                        node(200, emptyTree(), leaf(300)));
    cout << "8.21 handcrafted tree : " << shape(skew) << "\n";
    cout << "8.21 is it AVL?       : " << (isOrderStatisticTree(skew, why) ? "yes" : "no")
<< "\n";
    cout << "8.21 weight-balanced? : " << (isWeightBalanced(skew, why) ? "yes" : "no")
         << "   (fault: " << why << ")\n\n";

    cout << "8.22 insert 5, 15, 25 one at a time\n";
    OSTree u = t;
    for (int v : {5, 15, 25}) {
        rotations = 0; u = insertOS(v, u);
        cout << "      after " << setw(2) << v << ": rot=" << rotations
             << ", h=" << height(u) << ", n=" << sizeOf(u)
             << ", valid=" << (isOrderStatisticTree(u, why) ? "yes" : "NO") << "\n";
    }
    cout << "      shape: " << shape(u) << "\n\n";

    cout << "8.25 delete 50 (two children), then 10, then 999 (absent)\n";
    for (int v : {50, 10, 999}) {
        rotations = 0; u = deleteOS(u, v);
        cout << "      after " << setw(3) << v << ": rot=" << rotations
             << ", h=" << height(u) << ", n=" << sizeOf(u)
             << ", valid=" << (isOrderStatisticTree(u, why) ? "yes" : "NO") << "\n";
    }
    cout << "      shape: " << shape(u) << "\n\n";
```

Listing 8.12 (Continued) *Complete solutions to Problems 8.18 to 8.25, with a deliberate corruption of one stored field to show the validator of Problem 8.18 catching it, and a two-thousand-element stress test.*

```
    cout << "stress: insert 1..2000 ascending, select every 200th, delete all\n";
    rotations = 0;
    OSTree s = emptyTree();
    for (int v = 1; v <= 2000; ++v) s = insertOS(v, s);
    cout << "      n=" << sizeOf(s) << ", h=" << height(s)
         << ", rot=" << rotations
         << ", valid=" << (isOrderStatisticTree(s, why) ? "yes" : "NO") << "\n";
    cout << "      select(k) for k = 200,400,...,2000 :";
    for (int k = 200; k <= 2000; k += 200) cout << ' ' << select_(s, k);
    cout << "\n";
    for (int v = 1; v <= 2000; ++v) s = deleteOS(s, v);
    cout << "      after deleting all: n=" << sizeOf(s)
         << ", isEmpty=" << (isEmpty(s) ? "true" : "false") << "\n";
    return 0;
}
```

The program prints:

```
8.18 shape            :
50[11]{30[6]{20[2]{10[1],.},40[3]{35[1],45[1]}},70[4]{60[2]{.,65[1]},80[1]}}
8.18 valid OS tree    : yes   (fault: none)
8.18 after corrupting : no   (fault: node 50 stores size 99 but the truth is 11)
8.18 repaired         : yes

8.19 all values       : 10 20 30 35 40 45 50 60 65 70 80
8.19 in [30,60]       : 30 35 40 45 50 60
8.19 in [41,44]       : (nothing)

8.20 k-th smallest    : 10 20 30 35 40 45 50 60 65 70 80
8.20 select(6) = 45 after 4 node probes (n = 11, h = 3)

8.21 weight-balanced  : yes   (fault: none)
8.21 handcrafted tree :
100[10]{40[7]{20[3]{10[1],30[1]},60[3]{50[1],70[1]}},200[2]{.,300[1]}}
8.21 is it AVL?       : yes
8.21 weight-balanced? : no   (fault: node 100: left size 7, right size 2)

8.22 insert 5, 15, 25 one at a time
      after  5: rot=1, h=3, n=12, valid=yes
      after 15: rot=0, h=4, n=13, valid=yes
      after 25: rot=0, h=4, n=14, valid=yes
      shape:
50[14]{30[9]{10[5]{5[1],20[3]{15[1],25[1]}},40[3]{35[1],45[1]}},70[4]{60[2]{.,65[1]},80[1]}
}

8.25 delete 50 (two children), then 10, then 999 (absent)
      after  50: rot=0, h=4, n=13, valid=yes
      after  10: rot=1, h=4, n=12, valid=yes
      after 999: rot=0, h=4, n=12, valid=yes
      shape:
45[12]{30[7]{20[4]{5[2]{.,15[1]},25[1]},40[2]{35[1],.}},70[4]{60[2]{.,65[1]},80[1]}}

stress: insert 1..2000 ascending, select every 200th, delete all
      n=2000, h=10, rot=1989, valid=yes
      select(k) for k = 200,400,...,2000 : 200 400 600 800 1000 1200 1400 1600 1800 2000
      after deleting all: n=0, isEmpty=true
```

8.18.4 Reading the output as a check

Each block of the output answers one question about your own solution.

Problem 8.18. A correct validator says *no* to the corrupted tree and names node 50. A validator that recomputes sizes instead of comparing them against the stored field will report "yes" — it is validating nothing.

Problem 8.19. The three lines must read `10 20 30 35 40 45 50 60 65 70 80`, `30 35 40 45 50 60` and nothing at all. The pruning requirement is met only if the code has both guards; a version without them prints the same lines and visits every node.

Problem 8.20. `select(6) = 45` after **4** probes on a tree of height 3 — that is $h + 1$, one node per level. A probe count of 11 means the whole tree was traversed and the "one path" requirement is unmet.

Problem 8.21. The handcrafted tree must come back *AVL yes, weight-balanced no*, with the fault reported at node 100 as "left size 7, right size 2". If your answer to the one-sentence question was "yes, every AVL tree is weight-balanced", this output is the counterexample.

Problems 8.22 to 8.25. Every `valid=` must read `yes`, including after deleting the root and after deleting a value that is not present. In the stress test, two thousand ascending insertions must

give height 10, not 1999: the bound is $1.4405\lg 2002 - 1.3277 \approx 14.5$, and $10 \leq 14$. Deleting all two thousand must leave `n=0`.

> **WHY THESE PROBLEMS ARE HARDER THAN THEY LOOK.** The rotations of Problems 8.23 and 8.24 are character-for-character the ones in **Listing 8.8** — and yet they now also maintain a `size` field that Problems 8.8 and 8.9 never mention. That is not luck. It is the consequence of routing every construction through one function that fixes every derived field. The lesson is worth stating as a rule: *put the invariant repair in the constructor, and the algorithms stop caring how many invariants there are.*

8.19 Profitina in Practice

Profitina is an AI-visibility platform — GEO, *generative engine optimization* — running in production at `app.profitina.com`. It scans a brand's website, scores how ready that site is to be **cited by an AI answer engine** from real signals such as content extractability, structured data and technical authority, and then helps fix what it finds. It is a Python and FastAPI application on PostgreSQL; all of its AI work runs on local models through Ollama and Qwen2.5, so user data never leaves the server for AI processing. It keeps a weekly score history per brand, and a scheduler re-scans each site at a per-tier cadence.

The three chapters this one consolidates each have a place in that product. What follows describes the *shape* of each problem and the *class* of solution.

A scan queue is a priority queue. At any moment many sites are waiting for their next scan, each with a time at which it becomes due. The scheduler's question is always the same — *which scan is due first?* — and a max-heap keyed on urgency (or, equally, a min-heap keyed on due time) answers it in $O(1)$ and absorbs each new or rescheduled scan in $O(\lg n)$. When the queue has to be rebuilt from scratch, after a restart for instance, `BUILD-MAX-HEAP` does it in linear time — Problem 8.1, at scale.

Comparing two versions of a page is a sequence-alignment problem. A weekly score trend becomes useful when it can say *what changed* on the page between two scans. Lining up two versions of a text so that the unchanged parts match and the rest shows up as insertions and deletions is the problem family of Problem 8.17: the longest common subsequence is exactly the part that did not change. A table of $m \times n$ cells is affordable for a page's headings or paragraphs, not for its individual characters, so a practical comparison works on coarser units — the same trade of table size against detail that Chapter 7 discusses.

The tree problems of this chapter have a place too, and it is narrower and, in production, more valuable: **`isBST` and `isHeightBalanced` are not classroom exercises; they are the shape of a shipped feature.**

Every index in a long-running service must be able to audit itself. Profitina's scan scheduler keeps a due-time index — which scan is next, which are due between 02:00 and 04:00 tonight — and that index survives process restarts, staged releases, schema migrations and the occasional manual correction by an operator. Each of those is an opportunity for the index to stop being an index. A tree whose ordering has quietly broken does not crash; it returns *wrong answers quickly*, which is far worse, because nothing alerts. `isBST` and `isHeightBalanced` are the two functions that turn "the structure is probably fine" into "the structure was checked at 03:00 and here is the node that failed". They cost $\Theta(n)$ and they run when the system is idle.

Do not trust a stored field; recompute it. The warning of §8.9.7 — that a checker reading a cached height field is checking nothing — has a production counterpart. Profitina's staged release only reports success after the deployed version answers `/api/health`; a health check that merely reported the value of a variable set at start-up would confirm nothing at all. A real check does the work again and compares. That is exactly what Problem 8.18 demands, and exactly why **Listing 8.7** declares its node type without a height field.

A validator must name the fault, not just report one. "The index is corrupt" sends an engineer on a hunt. "Order PL-0431 at minute 500 is filed where only minutes below 20 belong" sends them to one row. The diagnostic variants in **Listing 8.4** and **Listing 8.7** cost four extra lines and turn a boolean into an incident report.

And recovery must not depend on the broken invariant. When an ordering has failed, a *search* through the tree is unreliable — but an inorder walk still visits every node, because traversal follows pointers, not comparisons. So the repair is always the same: walk the whole structure, collect the records, sort them by the real key, rebuild. **Listing 8.13** does precisely that.

> **TWO RULES WE DO NOT BEND.** First, the platform *advises*; the customer decides. A visibility score is a measurement with an explanation attached, never a promise about how an answer engine will behave. Second, anything touching personal data follows UU PDP 27/2022 — explicit consent, data export, complete account purge. A complete purge means every index holding a reference to that account must support deletion, which is one more practical reason Problem 8.10 matters: an append-only structure cannot be purged.

8.19.1 The worked case study: Profitina Laundry

Profitina Laundry is the teaching case study for this course — a small laundry in Keputih, Surabaya, not the real product.

The owner takes in orders through the day and promises each one a pickup time. She needs four things from her book: the earliest pickup still outstanding, the latest one, the queue in deadline order, and the ability to strike out an order when it is collected or add one when a customer walks in. That is exactly the set of operations Problems 8.5, 8.4, 8.7 and 8.10 provide, over a tree keyed by the pickup deadline in minutes after 07:00.

Listing 8.13 builds the index with `insertHB`, answers all four of her questions, and then does the thing that matters: somebody retypes a deadline *in place*, without re-filing the order, and the audit catches it, names it, and the index is rebuilt from an inorder walk. Prices are Rp 7.000 per kilogram, billed on whole kilograms with a two-kilogram minimum.

Listing 8.13 Profitina Laundry's pickup index, built entirely from the eight tree solutions of this chapter, with a self-audit that names the faulty record and a rebuild that survives a corrupted ordering.

```
// laundry.cpp -- Profitina Laundry: an auditable pickup index.
// Every routine used here is one of the tree solutions of this chapter.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

struct Order { int due; string id; int grams; };     // due = minutes after 07:00

struct LNode;
using Index = LNode*;
struct LNode { Order v; Index left, right; int h; };

Index  emptyIndex()      { return nullptr; }
bool   isEmpty(Index t)  { return t == nullptr; }
int    height(Index t)   { return isEmpty(t) ? -1 : t->h; }
int    key(Index t)      { return t->v.due; }
Index  leftOf(Index t)   { return t->left; }
Index  rightOf(Index t)  { return t->right; }
Index  node(const Order& v, Index l, Index r) {
    Index t = new LNode; t->v = v; t->left = l; t->right = r;
    t->h = 1 + max(height(l), height(r)); return t;
}
int balanceFactor(Index t) { return height(rightOf(t)) - height(leftOf(t)); }

// ---- Problems 8.8 and 8.9 ----
Index rotateRight(Index t) {
    Index l = leftOf(t);
    return node(l->v, leftOf(l), node(t->v, rightOf(l), rightOf(t)));
}
Index rotateLeft(Index t) {
    Index r = rightOf(t);
    return node(r->v, node(t->v, leftOf(t), leftOf(r)), rightOf(r));
}
Index rebalanceForLeft(Index t) {
    if (balanceFactor(t) >= -1) return t;
    Index l = leftOf(t);
    if (balanceFactor(l) <= 0) return rotateRight(t);
    return rotateRight(node(t->v, rotateLeft(l), rightOf(t)));
}
Index rebalanceForRight(Index t) {
    if (balanceFactor(t) <= 1) return t;
    Index r = rightOf(t);
    if (balanceFactor(r) >= 0) return rotateLeft(t);
```

Listing 8.13 (Continued) *Profitina Laundry's pickup index, built entirely from the eight tree solutions of this chapter, with a self-audit that names the faulty record and a rebuild that survives a corrupted ordering.*

```
    return rotateLeft(node(t->v, leftOf(t), rotateRight(r)));
}
// ---- Problem 8.7 ----
Index insertHB(const Order& o, Index t) {
    if (isEmpty(t)) return node(o, emptyIndex(), emptyIndex());
    if (o.due < key(t)) return rebalanceForLeft (node(t->v, insertHB(o, leftOf(t)),
rightOf(t)));
    if (o.due > key(t)) return rebalanceForRight(node(t->v, leftOf(t), insertHB(o,
rightOf(t))));
    return t;                                          // same minute: keep the first
}
// ---- Problem 8.5: the last pickup of the day, one path only ----
const Order& latest(Index t) { return isEmpty(rightOf(t)) ? t->v : latest(rightOf(t)); }
const Order& earliest(Index t){ return isEmpty(leftOf(t)) ? t->v : earliest(leftOf(t)); }
// ---- Problem 8.10 ----
Index deleteHB(Index t, int due) {
    if (isEmpty(t)) return t;
    if (due < key(t)) return rebalanceForRight(node(t->v, deleteHB(leftOf(t), due),
rightOf(t)));
    if (due > key(t)) return rebalanceForLeft (node(t->v, leftOf(t), deleteHB(rightOf(t),
due)));
    if (isEmpty(leftOf(t)))  return rightOf(t);
    if (isEmpty(rightOf(t))) return leftOf(t);
    Order m = latest(leftOf(t));
    return rebalanceForRight(node(m, deleteHB(leftOf(t), m.due), rightOf(t)));
}

// ---- Problem 8.3: is the index still a search tree? ----
static bool auditBST(Index t, long long lo, long long hi, string& why) {
    if (isEmpty(t)) return true;
    if (key(t) <= lo || key(t) >= hi) {
        why = "order " + t->v.id + " (minute " + to_string(key(t))
            + ") is filed where only minutes in (" + to_string(lo) + ","
            + to_string(hi) + ") belong";
        return false;
    }
    return auditBST(leftOf(t), lo, key(t), why) && auditBST(rightOf(t), key(t), hi, why);
}
// ---- Problem 8.6: is it still height-balanced? ----
static int auditHeight(Index t, string& why) {
    if (isEmpty(t)) return -1;
    int hl = auditHeight(leftOf(t),  why); if (hl == -2) return -2;
    int hr = auditHeight(rightOf(t), why); if (hr == -2) return -2;
    if (t->h != 1 + max(hl, hr)) {
        why = "order " + t->v.id + " stores height " + to_string(t->h)
            + " but the truth is " + to_string(1 + max(hl, hr));
        return -2;
    }
    if (abs(hr - hl) > 1) {
        why = "order " + t->v.id + " has balance factor " + to_string(hr - hl);
```

Listing 8.13 (Continued) *Profitina Laundry's pickup index, built entirely from the eight tree solutions of this chapter, with a self-audit that names the faulty record and a rebuild that survives a corrupted ordering.*

```
        return -2;
    }
    return 1 + max(hl, hr);
}
int countNodes(Index t) { return isEmpty(t) ? 0 : 1 + countNodes(leftOf(t)) +
countNodes(rightOf(t)); }

// An inorder walk reaches every node whatever the keys say, so it is the safe
// way to recover the contents of an index whose ordering has been corrupted.
void collectAll(Index t, vector<Order>& out) {
    if (isEmpty(t)) return;
    collectAll(leftOf(t), out);
    out.push_back(t->v);
    collectAll(rightOf(t), out);
}

string clockOf(int due) {                 // minutes after 07:00 -> HH:MM
    int m = 7 * 60 + due;
    char buf[16]; snprintf(buf, sizeof buf, "%02d:%02d", m / 60, m % 60);
    return string(buf);
}
string rupiah(long long v) {
    string s = to_string(v), out; int c = 0;
    for (int i = (int)s.size() - 1; i >= 0; --i) { out += s[i]; if (++c % 3 == 0 && i > 0)
out += '.'; }
    return "Rp " + string(out.rbegin(), out.rend());
}
// Rp 7.000 per kilogram, billed on whole kilograms, minimum two kilograms.
long long price(int grams) {
    int kg = (grams + 999) / 1000;
    if (kg < 2) kg = 2;
    return 7000LL * kg;
}
// ---- Problem 8.4: the queue, latest deadline first ----
void printLatestFirst(Index t) {
    if (isEmpty(t)) return;
    printLatestFirst(rightOf(t));
    cout << "    " << clockOf(key(t)) << "  " << setw(7) << left << t->v.id
         << right << setw(6) << t->v.grams << " g   " << rupiah(price(t->v.grams)) << "\n";
    printLatestFirst(leftOf(t));
}
void audit(const char* when, Index t) {
    string w1 = "-", w2 = "-";
    bool bst = auditBST(t, -1, 100000, w1);
    bool bal = auditHeight(t, w2) != -2;
    cout << "  audit " << when << ": orders=" << countNodes(t)
         << ", height=" << height(t)
         << ", search-tree=" << (bst ? "OK" : "BROKEN")
```

Listing 8.13 (Continued) *Profitina Laundry's pickup index, built entirely from the eight tree solutions of this chapter, with a self-audit that names the faulty record and a rebuild that survives a corrupted ordering.*

```
         << ", height-balanced=" << (bal ? "OK" : "BROKEN") << "\n";
    if (!bst) cout << "      " << w1 << "\n";
    if (!bal) cout << "      " << w2 << "\n";
}

int main() {
    vector<Order> book = {
        {  0, "PL-0431", 3200}, { 95, "PL-0432", 7400}, { 40, "PL-0433", 1500},
        {185, "PL-0434", 5000}, { 20, "PL-0435", 2600}, {260, "PL-0436", 12000},
        {130, "PL-0437",  900}, {305, "PL-0438", 4100}, { 75, "PL-0439", 6300},
        {215, "PL-0440", 2000}
    };
    Index idx = emptyIndex();
    for (const Order& o : book) idx = insertHB(o, idx);

    cout << "Profitina Laundry, Keputih -- pickup index for today\n";
    audit("after load", idx);
    cout << "  earliest pickup : " << clockOf(earliest(idx).due) << "  " <<
earliest(idx).id << "\n";
    cout << "  latest pickup   : " << clockOf(latest(idx).due)   << "  " << latest(idx).id
<< "\n";
    cout << "  queue, latest deadline first:\n";
    printLatestFirst(idx);

    long long total = 0;
    for (const Order& o : book) total += price(o.grams);
    cout << "  billed today    : " << rupiah(total) << "\n\n";

    cout << "PL-0437 collected at 09:10; PL-0433 collected at 07:40\n";
    idx = deleteHB(idx, 130);
    idx = deleteHB(idx, 40);
    audit("after two collections", idx);

    cout << "\nA walk-in at 11:00 with 8.5 kg, promised for 13:20 (minute 380)\n";
    idx = insertHB({380, "PL-0441", 8500}, idx);
    audit("after the walk-in", idx);
    cout << "  new latest pickup: " << clockOf(latest(idx).due) << "  " << latest(idx).id
<< "\n";

    cout << "\nNow a bad hand-edit: somebody retypes one deadline in place.\n";
    Index victim = idx;                     // find the leftmost node and move it late
    while (!isEmpty(leftOf(victim))) victim = leftOf(victim);
    cout << "  editing " << victim->v.id << " from " << clockOf(victim->v.due)
         << " to " << clockOf(500) << " without re-filing it\n";
    victim->v.due = 500;
    audit("after the bad edit", idx);
```

Listing 8.13 (Continued) *Profitina Laundry's pickup index, built entirely from the eight tree solutions of this chapter, with a self-audit that names the faulty record and a rebuild that survives a corrupted ordering.*

```
    cout << "\nRepair: read the index out in order and rebuild it with insertHB.\n";
    vector<Order> salvage;
    collectAll(idx, salvage);      // an inorder walk still reaches every order
    sort(salvage.begin(), salvage.end(), [](const Order& a, const Order& b){ return a.due <
b.due; });
    Index rebuilt = emptyIndex();
    for (const Order& o : salvage) rebuilt = insertHB(o, rebuilt);
    audit("after rebuild", rebuilt);
    cout << "  queue, latest deadline first:\n";
    printLatestFirst(rebuilt);
    return 0;
}
```

Its output:

```
Profitina Laundry, Keputih -- pickup index for today
  audit after load: orders=10, height=3, search-tree=OK, height-balanced=OK
  earliest pickup : 07:00  PL-0431
  latest pickup   : 12:05  PL-0438
  queue, latest deadline first:
    12:05  PL-0438  4100 g   Rp 35.000
    11:20  PL-0436 12000 g   Rp 84.000
    10:35  PL-0440  2000 g   Rp 14.000
    10:05  PL-0434  5000 g   Rp 35.000
    09:10  PL-0437   900 g   Rp 14.000
    08:35  PL-0432  7400 g   Rp 56.000
    08:15  PL-0439  6300 g   Rp 49.000
    07:40  PL-0433  1500 g   Rp 14.000
    07:20  PL-0435  2600 g   Rp 21.000
    07:00  PL-0431  3200 g   Rp 28.000
  billed today    : Rp 350.000

PL-0437 collected at 09:10; PL-0433 collected at 07:40
  audit after two collections: orders=8, height=3, search-tree=OK, height-balanced=OK

A walk-in at 11:00 with 8.5 kg, promised for 13:20 (minute 380)
  audit after the walk-in: orders=9, height=3, search-tree=OK, height-balanced=OK
  new latest pickup: 13:20  PL-0441

Now a bad hand-edit: somebody retypes one deadline in place.
  editing PL-0431 from 07:00 to 15:20 without re-filing it
  audit after the bad edit: orders=9, height=3, search-tree=BROKEN, height-balanced=OK
      order PL-0431 (minute 500) is filed where only minutes in (-1,20) belong

Repair: read the index out in order and rebuild it with insertHB.
  audit after rebuild: orders=9, height=3, search-tree=OK, height-balanced=OK
  queue, latest deadline first:
    15:20  PL-0431  3200 g   Rp 28.000
    13:20  PL-0441  8500 g   Rp 63.000
    12:05  PL-0438  4100 g   Rp 35.000
    11:20  PL-0436 12000 g   Rp 84.000
    10:35  PL-0440  2000 g   Rp 14.000
    10:05  PL-0434  5000 g   Rp 35.000
    08:35  PL-0432  7400 g   Rp 56.000
    08:15  PL-0439  6300 g   Rp 49.000
    07:20  PL-0435  2600 g   Rp 21.000
```

Four things to notice, each of which is one of this chapter's problems wearing an apron.

The billing arithmetic is worth checking by hand, because it is the kind of rule a real shop actually has. PL-0437 is 900 g — under one kilogram — but the minimum charge is two kilograms, so it bills at $2 \times$ Rp 7.000 = Rp 14.000, the same as PL-0440's exactly 2.000 g and PL-0433's 1.500 g rounded up to 2 kg. PL-0436 is 12.000 g, exactly 12 kg, so $12 \times$ Rp 7.000 = Rp 84.000. Adding the ten: $28 + 56 + 14 + 35 + 21 + 84 + 14 + 35 + 49 + 14$ = Rp 350.000 in thousands, which is what the program prints.

The audit is `OK` three times and then `BROKEN` once, and the broken line names the order and the interval it violated — "order PL-0431 (minute 500) is filed where only minutes in (-1,20) belong". The upper bound 20 came from PL-0435, two levels above. That is **Table 8.5**'s mechanism, in a shop.

`height-balanced` stays `OK` through the corruption. Of course it does: moving a value does not change the shape. This is the point of running *both* audits — each is blind to the other's failure mode, and a system that checks only one invariant is a system that believes half of what it should.

The rebuild recovers all nine orders. The inorder walk reaches every node whatever the keys say, and after sorting by the true deadline and reinserting with `insertHB`, the index is correct again — and PL-0431, whose deadline really did move to 15:20, is now properly the latest pickup of the day. The customer was late; the book now says so.

8.20 Common Mistakes, and How to Avoid Them

The lists at the end of each problem name the errors specific to it. These are the ones that cut across the whole chapter.

Reading the problem once. Most problems here carry a constraint that is not in the function's name: *recursive*, *without making a separate list*, *at most one path*, *must use the two helpers*, *one pass*. Each constraint is the real content of its problem, and each is easy to miss on a quick reading. Underline every constraint before you start.

Testing only the happy path. The empty tree, the single node, the two-node tree and the absent key find more bugs than any fifteen-node example. The tree programs of this chapter test them all; none of the bugs they find takes more than ten seconds to fix once found.

Trusting a stored field. A height cached in a node is a *claim*, not a fact. Validators must recompute; that is the whole point of **Listing 8.7** declaring its node type with no height field, and of Problem 8.18 in **Listing 8.12** comparing the stored size against the truth.

Checking locally what is true globally, or globally what is true locally. The BST property needs bounds; the heap property does not; red-black property 4 does not; red-black property 5 needs a value returned from below. Before writing any validator, ask: *is this a statement about a node and its children, or about a node and its whole subtree?*

Calling the rebalancer named after the side you recursed into. In insertion, yes. In deletion, **no** — the opposite. §8.13.3.

Asserting a complexity instead of measuring one. Every claim in this chapter is checked by a counter in a program: 4 083 swaps to build a heap of 4 095 keys in Problem 8.1, 522 753 comparisons against 8 194 for Quicksort's worst and best cases in Problem 8.2, 4 nodes against 15 in Problem 8.5, 769 against 127 in Problem 8.6, 3^{n-1} calls against $1 + (n^3 - n)/3$ in Problem 8.16, and 6 rotations on a height-12 Fibonacci tree in Exercise 7. Counters cost three lines and turn opinions into evidence.

Saying $O(\lg n)$ when you mean $O(h)$. True only with a balance guarantee. §8.8.5.

Heapifying or partitioning with the wrong loop bounds. `BUILD-MAX-HEAP` must count *down* from $\lfloor n/2 \rfloor$; Lomuto's loop runs j from p to $r - 1$ with i starting at $p - 1$; Quicksort recurses on $A[p..q-1]$ and $A[q+1..r]$. Each of these off-by-one errors produces a program that runs and is wrong. §8.3.9 and §8.4.9.

Filling a dynamic-programming table in the wrong order. A cell may only be computed after every cell it reads. For the matrix chain that means increasing chain length, not row by row; for the longest common subsequence, row by row works because every cell reads only the row above and the cell to its left. §8.15.12 and §8.16.9.

Keeping the value and throwing away the choice. $M[1,5] = 1\,344$ says what the best bracketing costs, not what it is; $c[6,5] = 4$ says how long the longest common subsequence is, not what it is. Store the split table and the arrow table, or the answer cannot be rebuilt.

8.21 Chapter Summary

- This chapter consolidates Chapters 5 to 7 through **twenty-five worked problems**, listed in **Table 8.1**, and draws on every earlier chapter, as **Table 8.2** shows. Almost every solution is one idea: carry information down a recursion or return it up, so that no work is done twice.
- **A multi-part problem is solved in dependency order**, not numerical order, and each part begins with five written items: the signature, the base case, the invariant, the smallest failing example and the complexity claim (§8.2).
- **Problem 8.1, `BUILD-MAX-HEAP`.** Heapify the internal nodes from $\lfloor n/2 \rfloor$ down to 1; on [5,13,2,25,7,17,20,8,4,11] that is 15 comparisons and 6 swaps (**Table 8.3**). The cost is $O(n)$ because most nodes are low (Equation (8.1)), and on a perfect heap the swaps never exceed $n - \lg(n+1)$. **Listing 8.1**.
- **Problem 8.2, Quicksort.** Lomuto's partition makes exactly $r - p$ comparisons and leaves the pivot in its final place (**Table 8.4**). Sorted input costs $\frac{n(n-1)}{2}$ comparisons (Equation (8.3)); perfectly balanced splits cost $(n+1)\lg(n+1) - 2n$ (Equation (8.4)); a random order lands close to the best case. **Listing 8.2**.
- **Problem 8.3, `isBST`.** The BST property is global, so a parent–child check is necessary but not sufficient; the fix is to carry an open interval (ℓ, u) down the recursion, splitting it at each node's value. Seeding `int` bounds with `INT_MIN` and `INT_MAX` is the tempting wrong solution — it rejects a legal tree containing either — and widening the bound type repairs it. The inorder walk is an independent second method. **Listing 8.4**.
- **Problem 8.4, `printDescending`.** Reverse inorder: right, node, left. Four lines, no list, $\Theta(n)$ time and $O(h)$ stack. **Listing 8.5**.
- **Problem 8.5, `max`.** Discard every left subtree unexamined and walk the right spine. $O(h)$, which is $O(\lg n)$ only because the tree is balanced — say so. **Listing 8.6**.
- **Problem 8.6, `isHeightBalanced`.** Return the height and fail with the sentinel -2, which is free because no real height is below -1. One pass, $\Theta(n)$, against $(n+1)\lg(n+1) - n$ for the recompute-everything version (Equation (8.10)) — a fact derived, then measured, then confirmed by the Master Theorem. **Listing 8.7**, **Table 8.6**.
- **Problems 8.7 to 8.10** are one design. `node()` is the sole constructor and fixes every derived field, so rotations are single expressions and stale heights are impossible. `insertHB` is ordinary BST insertion with a rebalancer wrapped around each rebuilt node; `rebalanceForLeft` is four cases collapsed to two; `rebalanceForRight` is its mirror; `deleteHB` calls the **opposite** rebalancer from the side it recursed into. The case test must be ≤ 0, not < 0, because deletion can present a level child. **Listing 8.8**, **Table 8.7**.
- **Problems 8.11 to 8.15, red-black trees**, have the same skeleton as the AVL problems: the same `isBST`, a local check that *is* correct (property 4), a one-pass sentinel check (black-height), an assembly, and two rotations — this time imperative, with six parent pointers to repair. **Listing 8.9**, **Table 8.8**, **Table 8.9**.
- **Problem 8.16, the matrix chain.** Filling M by increasing chain length, Equation (8.18) gives $M[1,5] = 1\,344$ for $d = \langle 4,10,3,12,20,7 \rangle$ with bracketing $((A_1A_2)((A_3A_4)A_5))$, against 1 784 left to right. Plain recursion makes 3^{n-1} calls; memoization makes $1 + (n^3 - n)/3$. **Listing 8.10**, **Table 8.10**, **Table 8.11**.
- **Problem 8.17, the longest common subsequence.** Filling c row by row with Equation (8.20) and walking the arrows back gives SORE for SORTED and STORE, in $\Theta(mn)$ time for the table and at most $m + n$ steps for the walk. **Listing 8.11**, **Table 8.12**.

- **Problems 8.18 to 8.25** repeat the tree problems on an order-statistic tree and show that a second derived field costs one line when every construction goes through one constructor. **Listing 8.12**, **Table 8.13**.
- Everything here is verified rather than asserted: every hand trace is reproduced by a program, every closed-form count is compared with a measured one, and the AVL routines are checked against all six worked examples of Chapter 6 and a thousand-element stress test against the height bound $1.4405\lg(n+2) - 1.3277$.

8.21.1 Looking ahead

Chapter 9 leaves trees for **greedy algorithms** — activity selection, Huffman coding and the matroid theory that says when greed is safe. The habit this chapter built, of asking *what invariant does my code maintain, and how would I check it?*, is the habit Chapter 9 needs most, because a greedy algorithm is correct exactly when a particular exchange invariant survives every step.

8.22 Exercises

1. Trace `isBST` on the tree $40\{20\{10,45\},60\{50,70\}\}$. Give the pair (ℓ, u) received by each call, in the order the calls are made, and say which node first violates its bounds.

2. What is the smallest number of nodes in a tree for which the local parent–child check passes but the tree is not a BST? Prove that two nodes can never do it.

3. Modify `printDescending` so that it prints only the k largest values and then stops, without visiting any other node. State its running time in terms of k and h.

4. Write `secondLargest(MyTree t)`, returning the second largest value of a BST in $O(h)$ time. Handle: the maximum has a left subtree; the maximum is a leaf; the root itself is the maximum; the tree has fewer than two values.

5. Compute, by the closed form of §8.9.3, the number of node inspections `isHeightBalancedSlow` performs on a perfect tree of height 4, and check it against **Listing 8.14**.

6. Insert 1,2,3,4,5,6,7 in that order into an empty AVL tree using `insertHB`. Give the shape and the rotation count after each insertion, and name the case for each rotation.

7. Build the minimum-node AVL tree of height 6 — the Fibonacci tree $N(6)$. How many nodes does it have? Delete its largest key and count the rebalancings. Then say how the count grows with h.

8. Construct a nine-node AVL tree and a deletion from it such that `rebalanceForLeft` written with `< 0` instead of `<= 0` produces a tree that is not an AVL tree. Give both results.

9. For the red-black tree of **Listing 8.9** — root 26:B, with 17:R holding 14:B and 21:B, and 41:B holding 47:R — compute `countBlackNodes`, the black-height of the root, and the black-height of node 41. Then recolor 41 red and say which property breaks.

10. Solve $T(n) = 2T(n/2) + \Theta(n)$ and $T(n) = 2T(n/2) + \Theta(1)$ with the Master Theorem, naming the case in each, and say which routine of Problem 8.6 each describes.

11. Run `BUILD-MAX-HEAP` by hand on $A = [1,2,3,4,5,6,7]$. Give the array after each call of `MAX-HEAPIFY`, count the comparisons and swaps, and compare the swap count with the bound $n - \lg(n+1)$ of §8.3.6.

12. Apply `PARTITION(A, 1, 5)` to $A = [5,5,5,5,5]$. Where does the pivot end up? How many comparisons does the complete `QUICKSORT` make on n equal keys, and what does that say about inputs with many repeated values?

13. Solve the matrix chain $d = \langle 5,4,6,2,7 \rangle$ by hand: fill M and c, give the least cost and the optimal bracketing, and compare it with the left-to-right order.

14. Find a longest common subsequence of $X = $ LAUNDRY and $Y = $ SUNDAY by filling the table of **Pseudocode 8.6** and walking the arrows back. Is it the only one?

8.23 Solutions to the Exercises

1. Table 8.14 lists the calls, in order, with the bounds each receives.

Table 8.14 Exercise 1: the recursion stops at node 45, whose upper bound 40 came from the root two levels above. Nodes 60, 50 and 70 are never visited.

Call	Node	ℓ	u	Verdict
1	40	$-\infty$	$+\infty$	inside; recurse
2	20	$-\infty$	40	inside; recurse
3	10	$-\infty$	20	inside; leaf; **true**
4	45	20	40	$45 \geq 40$ — **violates the upper bound**

Node 45 is the first violation. The local check would have passed everywhere: $10 < 20 < 45$ at node 20, and $20 < 40 < 60$ at the root.

2. Three nodes suffice, as §8.6.3 showed: root 20, left child 10, and 10's right child 25. Local checks pass ($10 < 20$ and $25 > 10$) but $25 > 20$ lies in the left subtree.

Two nodes cannot. With two nodes there is a root and one child, and the child is either the root's left child or its right child. The BST property for such a tree says exactly one thing — *the left child is smaller* or *the right child is larger* — and that is precisely the local check. There is no ancestor-versus-grandchild relation to miss, because there is no grandchild. Hence the local check is equivalent to the BST property on every tree of at most two nodes, and three is the minimum. (One node and zero nodes are trivially the same.)

3.

```
void printKLargest(MyTree t, int& k) {
    if (isEmpty(t) || k == 0) return;      // stop as soon as the quota is met
    printKLargest(t->right, k);
    if (k == 0) return;
    cout << t->value << ' '; --k;
    printKLargest(t->left, k);
}
```

The counter `k` must be shared across the whole recursion — a reference in C++ — for the reason given in §8.6.8. Running time: the walk descends the right spine once, costing $O(h)$, and thereafter each of the k values printed costs $O(1)$ amortized plus at most one extra descent, giving $O(h + k)$. Space $O(h)$.

4. The key observation: the maximum is at the end of the right spine. Its predecessor — the second largest — is either the maximum of *its* left subtree, if it has one, or else its parent. Three cases, one path, $O(h)$. `secondLargest` in **Listing 8.14** is the code. Its output covers all three shapes:

```
=== Exercise 4: second largest, in O(h) ===
  50(0){30(0){20(0),40(0)},70(0){60(0),80(0)}}
  largest = 80, second largest = 70
  50(1){30(0),70(-1){65(0),.}}
  largest = 70, second largest = 65
  50(-1){30(0),.}
  largest = 50, second largest = 30
```

Line 2: the maximum 80 is a leaf, so the answer is its parent, 70. Line 4: the maximum 70 has a left child, so the answer is the maximum of that subtree, 65. Line 6: the root 50 is the maximum, so the answer is the maximum of its left subtree, 30.

5. A perfect tree of height $k = 4$ has $n = 2^5 - 1 = 31$ nodes. The closed form gives

$$W = (n+1)\lg(n+1) - n = 32 \times 5 - 31 = 160 - 31 = 129.$$

Listing 8.14 prints `31 129 129 31 match`. The one-pass method inspects 31 nodes, one per node, as it must.

6. Table 8.15 gives the answer, produced by the same `insertHB` as **Listing 8.8**.

Table 8.15 Exercise 6: inserting 1 to 7 in ascending order. Every rotation is the Right-Right case, because ascending input always leans right, and four single rotations are enough to keep the height at 2 instead of 6.

Insert	Rotations	Case	Shape, with balance factors
1	0	—	`1(0)`
2	0	—	`1(1){.,2(0)}`
3	1	RR	`2(0){1(0),3(0)}`
4	0	—	`2(1){1(0),3(1){.,4(0)}}`
5	1	RR	`2(1){1(0),4(0){3(0),5(0)}}`
6	1	RR	`4(0){2(0){1(0),3(0)},5(1){.,6(0)}}`
7	1	RR	`4(0){2(0){1(0),3(0)},6(0){5(0),7(0)}}`

7. $N(h) = 1 + N(h-1) + N(h-2)$ with $N(0) = 1$ and $N(1) = 2$ gives $N(2) = 4$, $N(3) = 7$, $N(4) = 12$, $N(5) = 20$, $N(6) = \mathbf{33}$ — matching $N(h) = F(h+3) - 1$, since $F(9) = 34$. **Listing 8.14** prints for $h = 6$: 33 nodes, **3 rotations**, height falling from 6 to 5, tree still AVL. The table it prints for $h = 2$ to 12 gives rotation counts 1,1,2,2,3,3,4,4,5,5,6 — that is $\lfloor h/2 \rfloor$, and since $h = \Theta(\lg n)$ the number of rebalancings caused by one deletion is $\Theta(\lg n)$. Contrast the at-most-one rebalancing of an insertion (§8.13.4).

8. The nine-node tree

$$50\{\,20\{10\{5,15\},\ 30\{.,40\}\},\ 70\{60,.\}\,\}$$

is an AVL tree: $\mathrm{bf}(10) = 0$, $\mathrm{bf}(30) = +1$, $\mathrm{bf}(20) = 0$, $\mathrm{bf}(70) = -1$, $\mathrm{bf}(50) = 1 - 2 = -1$. Delete 60. Node 70 becomes a leaf, so at node 50 we get $\mathrm{bf}(50) = 0 - 2 = -2$ with $\mathrm{bf}(20) = 0$ — the level child that only deletion can produce. **Listing 8.14** runs both versions:

```
=== Exercise 8: why the case test needs <= and not < ===
  start          : 50(-1){20(0){10(0){5(0),15(0)},30(1){.,40(0)}},70(-1){60(0),.}}
AVL=yes
  delete 60, <=0 : 20(1){10(0){5(0),15(0)},50(-1){30(1){.,40(0)},70(0)}}   AVL=yes
  delete 60, <0  : 30(-1){20(-2){10(0){5(0),15(0)},.},50(0){40(0),70(0)}}   AVL=NO
```

With `<= 0` the LL branch performs one right rotation and the result is a legal AVL tree. With `< 0` the LR branch performs a double rotation, and node 20 ends with balance factor -2: the result is a binary search tree with the right values in the right order, and it is not height-balanced. This is the bug §8.11.5 predicted, exhibited on the smallest tree that shows it.

9. Black nodes: 26, 14, 21, 41 — **four**. Black-height of the root: every path to a `NIL` passes two black nodes not counting the `NIL`; for instance 26 → 17 → 14 → NIL counts 26 and 14, and 26 → 41 → 47 → NIL counts 26 and 41. So **2**. Black-height of node 41: paths 41 → NIL and 41 → 47 → NIL each count 41 alone, so **1**. **Listing 8.9** prints `black-height of the root = 2`.

Recolor 41 red. Property 4 is still satisfied — 41's children are the black `NIL` and the red 47 — no: 47 *is* red, so a red 41 with a red child 47 breaks **property 4** immediately. Property 5 also breaks: the path 26 → 41 → NIL now counts only 26, giving black-height 1 on that side against 2 on the left. Both `validateProperty4` and `validateBlackHeight` reject it.

10. $T(n) = 2T(n/2) + \Theta(n)$: $a = 2$, $b = 2$, $\log_b a = 1$, $n^{\log_b a} = n$, and $f(n) = \Theta(n) = \Theta(n^{\log_b a})$ — **case 2** — so $T(n) = \Theta(n\lg n)$. This is `isHeightBalancedSlow`, which pays $\Theta(n)$ at each node to recompute heights.

$T(n) = 2T(n/2) + \Theta(1)$: same a, b and $\log_b a$, but $f(n) = \Theta(1) = O(n^{1-\varepsilon})$ with $\varepsilon = 1$ — **case 1** — so $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$. This is `checkedHeight`, the one-pass sentinel method.

11. Here $n = 7$, so the loop runs $i = 3,2,1$.

- $i = 3$: node 3 holds 3, with children $A[6] = 6$ and $A[7] = 7$; the larger, 7, exceeds 3, so swap: [1,2,7,4,5,6,3]. Node 7 is a leaf. (2 comparisons, 1 swap.)
- $i = 2$: node 2 holds 2, with children 4 and 5; swap with 5: [1,5,7,4,2,6,3]. Node 5 is a leaf. (2 comparisons, 1 swap.)
- $i = 1$: the root 1 has children 5 and 7; swap with 7: [7,5,1,4,2,6,3]. At node 3 the value 1 has children 6 and 3; swap with 6: [7,5,6,4,2,1,3]. Node 6 is a leaf. (4 comparisons, 2 swaps.)

The heap is [7,5,6,4,2,1,3], after **8 comparisons and 4 swaps**. The bound is $n - \lg(n+1) = 7 - 3 = 4$, so ascending input reaches it exactly: every value sank as far as it could. This is the first row of the measurement table printed by **Listing 8.1**.

12. Every $A[j] = 5 \le 5$, so i advances at every step and every exchange is of an element with itself; line 7 exchanges $A[5]$ with itself and returns $q = 5$. The pivot lands at the far right, and the split is $n - 1$ against 0 — the worst case of §8.4.5. The recursion repeats this on $n - 1$ equal keys, then $n - 2$, and so on, so the complete sort makes

$$C(n) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$$

comparisons: 10 for $n = 5$, and $\Theta(n^2)$ in general. Lomuto's partition therefore degrades on inputs with many repeated values — a real concern, since a column of prices or dates often contains long runs of equal entries. Hoare's partition of Chapter 5, which stops on equal keys from both sides and so splits a run of equal keys near the middle, or a three-way partition that sets all keys equal to the pivot aside at once, avoids the problem.

13. $d_0, \ldots, d_4 = 5,4,6,2,7$.

- Length 2: $M[1,2] = 5 \cdot 4 \cdot 6 = 120$; $M[2,3] = 4 \cdot 6 \cdot 2 = 48$; $M[3,4] = 6 \cdot 2 \cdot 7 = 84$.
- $M[1,3] = \min(0 + 48 + 5 \cdot 4 \cdot 2,\ 120 + 0 + 5 \cdot 6 \cdot 2) = \min(88,180) = 88$, with $c[1,3] = 1$.
- $M[2,4] = \min(0 + 84 + 4 \cdot 6 \cdot 7,\ 48 + 0 + 4 \cdot 2 \cdot 7) = \min(252,104) = 104$, with $c[2,4] = 3$.

- $M[1,4] = \min(0 + 104 + 5 \cdot 4 \cdot 7,\ 120 + 84 + 5 \cdot 6 \cdot 7,\ 88 + 0 + 5 \cdot 2 \cdot 7) = \min(244,414,158) = 158$, with $c[1,4] = 3$.

The least cost is 158. Reading c: $c[1,4] = 3$ splits off A_4; $c[1,3] = 1$ splits A_1 from A_2A_3. The bracketing is $((A_1(A_2A_3))A_4)$: first A_2A_3 for 48, giving 4×2; then A_1 times that for $5 \cdot 4 \cdot 2 = 40$, giving 5×2; then times A_4 for $5 \cdot 2 \cdot 7 = 70$. Total $48 + 40 + 70 = 158$ ✓. Left to right costs $120 + 60 + 70 = 250$. The winning order reaches the small dimension 2 first and keeps it.

14. With $X = $ L, A, U, N, D, R, Y (rows) and $Y = $ S, U, N, D, A, Y (columns), the rows of the table are:

- L: 0 ↑, 0 ↑, 0 ↑, 0 ↑, 0 ↑, 0 ↑ (L is not in SUNDAY; every tie goes up)
- A: 0 ↑, 0 ↑, 0 ↑, 0 ↑, 1 ↖, 1 ←
- U: 0 ↑, 1 ↖, 1 ←, 1 ←, 1 ↑, 1 ↑
- N: 0 ↑, 1 ↑, 2 ↖, 2 ←, 2 ←, 2 ←
- D: 0 ↑, 1 ↑, 2 ↑, 3 ↖, 3 ←, 3 ←
- R: 0 ↑, 1 ↑, 2 ↑, 3 ↑, 3 ↑, 3 ↑
- Y: 0 ↑, 1 ↑, 2 ↑, 3 ↑, 3 ↑, 4 ↖

The length is $c[7,6] = 4$. The walk back: (7,6) ↖, take Y; (6,5) ↑; (5,5) ←; (5,4) ↖, take D; (4,3) ↖, take N; (3,2) ↖, take U; (2,1) ↑; (1,1) ↑; stop. Reversed: **UNDY**. It is the only one of length 4: the only other shared letter, A, comes *before* U, N and D in LAUNDRY but *after* them in SUNDAY, so it can join none of them, and a common subsequence containing A has length at most 2 (AY).

Listing 8.14 Runnable solutions to exercises 4, 5, 7 and 8, with a `STRICT` switch that reproduces the `< 0` bug of exercise 8 on demand.

```
// exercises.cpp -- runnable solutions to exercises 4, 5 and 7 of chapter 8.
#include <iostream>
#include <iomanip>
#include <string>
#include <stdexcept>
#include <vector>
#include <cmath>
#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; }
int    value(MyTree t)   { return t->value; }
MyTree leftOf(MyTree t)  { return t->left; }
MyTree rightOf(MyTree t) { return t->right; }
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(rightOf(t)) - height(leftOf(t)); }

long long rotations = 0;
bool STRICT = false;        // Exercise 8: true makes the case test use < instead of <=
MyTree rotateRight(MyTree t) { ++rotations; MyTree l = leftOf(t);
    return node(value(l), leftOf(l), node(value(t), rightOf(l), rightOf(t))); }
MyTree rotateLeft(MyTree t)  { ++rotations; MyTree r = rightOf(t);
    return node(value(r), node(value(t), leftOf(t), leftOf(r)), rightOf(r)); }
MyTree rebalanceForLeft(MyTree t) {
    if (balanceFactor(t) >= -1) return t;
    MyTree l = leftOf(t);
    bool single = STRICT ? (balanceFactor(l) < 0) : (balanceFactor(l) <= 0);
    if (single) return rotateRight(t);
    return rotateRight(node(value(t), rotateLeft(l), rightOf(t)));
}
MyTree rebalanceForRight(MyTree t) {
    if (balanceFactor(t) <= 1) return t;
    MyTree r = rightOf(t);
    bool single = STRICT ? (balanceFactor(r) > 0) : (balanceFactor(r) >= 0);
    if (single) return rotateLeft(t);
    return rotateLeft(node(value(t), leftOf(t), rotateRight(r)));
```

Listing 8.14 (Continued) *Runnable solutions to exercises 4, 5, 7 and 8, with a STRICT switch that reproduces the < 0 bug of exercise 8 on demand.*

```
}
MyTree insertHB(int n, MyTree t) {
    if (isEmpty(t)) return node(n, emptyTree(), emptyTree());
    if (n < value(t)) return rebalanceForLeft (node(value(t), insertHB(n, leftOf(t)),
rightOf(t)));
    if (n > value(t)) return rebalanceForRight(node(value(t), leftOf(t), insertHB(n,
rightOf(t))));
    return t;
}
int maxOf(MyTree t) { return isEmpty(rightOf(t)) ? value(t) : maxOf(rightOf(t)); }
MyTree deleteHB(MyTree t, int x) {
    if (isEmpty(t)) return t;
    if (x < value(t)) return rebalanceForRight(node(value(t), deleteHB(leftOf(t), x),
rightOf(t)));
    if (x > value(t)) return rebalanceForLeft (node(value(t), leftOf(t),
deleteHB(rightOf(t), x)));
    if (isEmpty(leftOf(t)))  return rightOf(t);
    if (isEmpty(rightOf(t))) return leftOf(t);
    int m = maxOf(leftOf(t));
    return rebalanceForRight(node(m, deleteHB(leftOf(t), m), rightOf(t)));
}
string shape(MyTree t) {
    if (isEmpty(t)) return ".";
    string s = to_string(value(t)) + "(" + to_string(balanceFactor(t)) + ")";
    if (isEmpty(leftOf(t)) && isEmpty(rightOf(t))) return s;
    return s + "{" + shape(leftOf(t)) + "," + shape(rightOf(t)) + "}";
}
int countNodes(MyTree t) { return isEmpty(t) ? 0 : 1 + countNodes(leftOf(t)) +
countNodes(rightOf(t)); }
bool isAVL(MyTree t) {
    if (isEmpty(t)) return true;
    return abs(balanceFactor(t)) <= 1 && isAVL(leftOf(t)) && isAVL(rightOf(t));
}

// ===== Exercise 4: the second largest value of a BST, in O(h) =====
// The largest value is the end of the right spine. The second largest is either
// the parent of that node, or the largest value in that node's LEFT subtree.
int secondLargest(MyTree t) {
    if (isEmpty(t)) throw invalid_argument("the tree is empty");
    if (isEmpty(rightOf(t))) {                        // the root IS the largest
        if (isEmpty(leftOf(t))) throw invalid_argument("fewer than two values");
        return maxOf(leftOf(t));                      // largest of the left subtree
    }
    MyTree parent = t, cur = rightOf(t);
    while (!isEmpty(rightOf(cur))) { parent = cur; cur = rightOf(cur); }
    // cur holds the maximum; parent is its parent
    if (!isEmpty(leftOf(cur))) return maxOf(leftOf(cur));
    return value(parent);
}
```

Listing 8.14 (Continued) *Runnable solutions to exercises 4, 5, 7 and 8, with a STRICT switch that reproduces the < 0 bug of exercise 8 on demand.*

```
// ===== Exercise 7: a minimum-node AVL tree of height h (a Fibonacci tree) =====
// N(-1)=0, N(0)=1, N(h)=1+N(h-1)+N(h-2). Keys are handed out in inorder, so
// the result is a genuine BST.
int nextKey;
MyTree fibTree(int h) {
    if (h < 0) return emptyTree();
    if (h == 0) return node(nextKey++, emptyTree(), emptyTree());
    MyTree l = fibTree(h - 1);
    int v = nextKey++;
    MyTree r = fibTree(h - 2);
    return node(v, l, r);
}

// ===== Exercise 5: the exact cost of the two balance checks =====
long long work = 0;
int heightSlow(MyTree t) {
    if (isEmpty(t)) return -1;
    ++work;
    return 1 + max(heightSlow(leftOf(t)), heightSlow(rightOf(t)));
}
bool isHBSlow(MyTree t) {
    if (isEmpty(t)) return true;
    ++work;
    if (abs(heightSlow(leftOf(t)) - heightSlow(rightOf(t))) > 1) return false;
    return isHBSlow(leftOf(t)) && isHBSlow(rightOf(t));
}
int checkedHeight(MyTree t) {
    if (isEmpty(t)) return -1;
    ++work;
    int hl = checkedHeight(leftOf(t));  if (hl == -2) return -2;
    int hr = checkedHeight(rightOf(t)); if (hr == -2) return -2;
    if (abs(hl - hr) > 1) return -2;
    return 1 + max(hl, hr);
}
MyTree perfect(int lo, int hi) {
    if (lo > hi) return emptyTree();
    int mid = lo + (hi - lo) / 2;
    return node(mid, perfect(lo, mid - 1), perfect(mid + 1, hi));
}
```

Listing 8.14 (Continued) *Runnable solutions to exercises 4, 5, 7 and 8, with a STRICT switch that reproduces the < 0 bug of exercise 8 on demand.*

```
int main() {
    cout << "=== Exercise 4: second largest, in O(h) ===\n";
    {
        MyTree t = emptyTree();
        for (int v : {50, 30, 70, 20, 40, 60, 80}) t = insertHB(v, t);
        cout << "  " << shape(t) << "\n";
        cout << "  largest = " << maxOf(t) << ", second largest = " << secondLargest(t) <<
"\n";
        MyTree u = emptyTree();
        for (int v : {50, 30, 70, 65}) u = insertHB(v, u);      // max has a left child
        cout << "  " << shape(u) << "\n";
        cout << "  largest = " << maxOf(u) << ", second largest = " << secondLargest(u) <<
"\n";
        MyTree w = emptyTree();
        for (int v : {50, 30}) w = insertHB(v, w);              // the root is the max
        cout << "  " << shape(w) << "\n";
        cout << "  largest = " << maxOf(w) << ", second largest = " << secondLargest(w) <<
"\n";
    }

    cout << "\n=== Exercise 8: why the case test needs <= and not < ===\n";
    {
        MyTree base = node(50,
                           node(20, node(10, leaf(5), leaf(15)), node(30, emptyTree(),
leaf(40))),
                           node(70, leaf(60), emptyTree()));
        cout << "  start          : " << shape(base)
             << "   AVL=" << (isAVL(base) ? "yes" : "NO") << "\n";
        STRICT = false; MyTree a = deleteHB(base, 60);
        cout << "  delete 60, <=0 : " << shape(a) << "   AVL=" << (isAVL(a) ? "yes" : "NO")
<< "\n";
        STRICT = true;  MyTree b = deleteHB(base, 60);
        cout << "  delete 60, <0  : " << shape(b) << "   AVL=" << (isAVL(b) ? "yes" : "NO")
<< "\n";
        STRICT = false;
    }

    cout << "\n=== Exercise 7: one deletion, many rebalancings ===\n";
    cout << "  h   N(h)   deleting the largest key: rotations\n";
    for (int h = 2; h <= 12; ++h) {
        nextKey = 1;
        MyTree f = fibTree(h);
        int n = countNodes(f);
        rotations = 0;
        MyTree g = deleteHB(f, n);              // n is the largest key by construction
        cout << "  " << setw(2) << h << setw(7) << n << setw(28) << rotations
             << "   (still AVL: " << (isAVL(g) ? "yes" : "no")
             << ", height " << height(f) << " -> " << height(g) << ")\n";
    }
```

Listing 8.14 (Continued) *Runnable solutions to exercises 4, 5, 7 and 8, with a STRICT switch that reproduces the < 0 bug of exercise 8 on demand.*

```
    cout << "\n=== Exercise 5: node inspections on a perfect tree ===\n";
    cout << "     n      slow    (n+1)lg(n+1)-n     fast\n";
    for (int k = 2; k <= 10; ++k) {
        int n = (1 << (k + 1)) - 1;
        MyTree p = perfect(1, n);
        work = 0; isHBSlow(p);        long long ws = work;
        work = 0; checkedHeight(p);   long long wf = work;
        long long closed = (long long)(n + 1) * (k + 1) - n;
        cout << setw(6) << n << setw(10) << ws << setw(18) << closed << setw(9) << wf
             << (ws == closed ? "   match" : "   MISMATCH") << "\n";
    }
    return 0;
}
```

Its full output:

```
=== Exercise 4: second largest, in O(h) ===
  50(0){30(0){20(0),40(0)},70(0){60(0),80(0)}}
  largest = 80, second largest = 70
  50(1){30(0),70(-1){65(0),.}}
  largest = 70, second largest = 65
  50(-1){30(0),.}
  largest = 50, second largest = 30

=== Exercise 8: why the case test needs <= and not < ===
  start          : 50(-1){20(0){10(0){5(0),15(0)},30(1){.,40(0)}},70(-1){60(0),.}}
AVL=yes
  delete 60, <=0 : 20(1){10(0){5(0),15(0)},50(-1){30(1){.,40(0)},70(0)}}   AVL=yes
  delete 60, <0  : 30(-1){20(-2){10(0){5(0),15(0)},.},50(0){40(0),70(0)}}   AVL=NO

=== Exercise 7: one deletion, many rebalancings ===
  h   N(h)   deleting the largest key: rotations
   2      4                           1   (still AVL: yes, height 2 -> 1)
   3      7                           1   (still AVL: yes, height 3 -> 2)
   4     12                           2   (still AVL: yes, height 4 -> 3)
   5     20                           2   (still AVL: yes, height 5 -> 4)
   6     33                           3   (still AVL: yes, height 6 -> 5)
   7     54                           3   (still AVL: yes, height 7 -> 6)
   8     88                           4   (still AVL: yes, height 8 -> 7)
   9    143                           4   (still AVL: yes, height 9 -> 8)
  10    232                           5   (still AVL: yes, height 10 -> 9)
  11    376                           5   (still AVL: yes, height 11 -> 10)
  12    609                           6   (still AVL: yes, height 12 -> 11)

=== Exercise 5: node inspections on a perfect tree ===
     n      slow    (n+1)lg(n+1)-n     fast
     7        17                17        7   match
    15        49                49       15   match
    31       129               129       31   match
    63       321               321       63   match
   127       769               769      127   match
   255      1793              1793      255   match
   511      4097              4097      511   match
  1023      9217              9217     1023   match
  2047     20481             20481     2047   match
```

The last block is the closed form of §8.9.3 checked against the machine at nine different sizes, and it matches at every one.

References

[1] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. 19th Annual Symposium on Foundations of Computer Science (FOCS)*, Ann Arbor, MI, USA, 1978, pp. 8–21.

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

[3] J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, 1964.

[4] R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, vol. 7, no. 12, p. 701, 1964.

[5] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[6] G. M. Adelson-Velsky and E. M. Landis, "An algorithm for the organization of information," *Doklady Akademii Nauk SSSR*, vol. 146, no. 2, pp. 263–266, 1962. (English translation: *Soviet Mathematics Doklady*, vol. 3, pp. 1259–1263, 1962.)

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

[8] N. Wirth, *Algorithms + Data Structures = Programs*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1976.

[9] C. Okasaki, *Purely Functional Data Structures*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

[10] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton Univ. Press, 1957.

[11] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974.

[12] M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, 4th ed. Boston, MA, USA: Pearson, 2014.

[13] ISO/IEC 14882:2024, *Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2024.

The heap procedures of Problem 8.1 and the partition and analysis of Problem 8.2 follow Cormen et al. [2, chs. 6–7]; heapsort is due to Williams [3], the bottom-up linear-time construction to Floyd [4], and Quicksort to Hoare [5], whose paper also introduced the two-pointer partition of Chapter 5. The AVL invariant and the height bound $1.4405\lg(n+2) - 1.3277$ are due to [6] and are analyzed in [7, §6.2.3]; heapsort is analyzed in [7, §5.2.3]. The *functional* formulation used for the tree problems — a single constructor, rotations as expressions, `insertHB` and `deleteHB` returning rebuilt trees — is the style of [8] and, in its modern form, [9]; it is why the height-refresh bug of Chapter 6 cannot occur here. Red-black trees in their colored form are due to Guibas and Sedgewick [1], and the imperative rotations with parent pointers of Problem 8.15 follow [2, ch. 13]. Dynamic programming as a method is due to Bellman [10]; the matrix-chain and longest-common-subsequence algorithms of Problems 8.16 and 8.17 follow [2, ch. 14], and the table-filling approach to comparing two strings goes back to Wagner and Fischer [11]. Practical guidance on balanced-tree implementation is in [12, ch. 4], and the language rules every listing obeys are [13].

Chapter 9
Greedy Algorithms

By the end of this chapter you will be able to look at an optimization problem and say whether taking the obviously attractive choice, right now, without thinking about the future, happens to be safe. You will be able to write down a greedy algorithm in three lines, and — far more important — to *prove* that it is correct, using the exchange argument, which is the single most valuable proof technique in this book. You will be able to recognize the two ingredients a problem must have before greed works: the greedy-choice property and optimal substructure. You will be able to solve the activity-selection problem recursively and iteratively in $\Theta(n\lg n)$ and prove both forms optimal. You will be able to say exactly why greed solves the fractional knapsack and ruins the 0-1 knapsack, with the numbers in front of you. You will be able to build a Huffman code by hand and in C++, encode a file, decode it back, and prove the code optimal. You will know what a matroid is, why matroids are the deepest answer we have to the question "when does greed work?", and how to schedule tasks with deadlines and penalties using one. And you will know, when you meet Kruskal, Prim and Dijkstra in later chapters, that you have already seen the shape of their correctness proofs.

9.1 Where We Left Off

Chapter 7 taught us dynamic programming. It gave us a recipe with four steps (§7.8.4): characterize the structure of an optimal solution, define its value recursively, compute that value in an order that respects the dependencies, and construct the solution from a table of recorded choices. It gave us the property that licenses the whole enterprise, **optimal substructure** (§7.5.2): *an optimal solution to a problem contains within it optimal solutions to the subproblems it is composed of.* And it gave us the standard way to prove that property, the **cut-and-paste argument** (§7.5.3): assume a piece of an optimal solution is not optimal for its own subproblem, cut it out, paste a better piece in, check the result is still legal, and observe that you have beaten something you assumed was unbeatable.

Chapter 7 also taught us dynamic programming's cost. The matrix-chain algorithm of §7.6.7 fills a table of $\Theta(n^2)$ cells, and computing each cell means taking a minimum over up to n candidate split points, so the whole thing is $\Theta(n^3)$ — exactly $(n^3 - n)/6$ innermost iterations, as §7.6.10 counted. The longest-common-subsequence algorithm of §7.9.6 is $\Theta(mn)$ in time and, without the space-saving trick, $\Theta(mn)$ in memory too; at $m = n = 8000$ that was 488 MiB of table.

Why so expensive? Because at every cell the dynamic program *does not know which choice is right*, so it tries them all. In matrix-chain multiplication it tries every split point k between i and j. In the longest common subsequence it tries dropping the last character of X and it tries dropping the last character of Y, and takes whichever turns out better. The table exists precisely because the choice cannot be made without first solving the subproblems.

9.1.1 What if the choice were obvious?

Now suppose that at each step there were no choice to agonize over. Suppose that one of the candidates were *visibly* the right one — right without solving any subproblem, right by inspection,

right for a reason you could state in a sentence. Then you would not need the table. You would take that candidate, reduce the problem to a single smaller problem, take the visibly right candidate there, and continue until nothing was left. No minimization over k. No $\Theta(n^2)$ cells. Often no memory at all beyond a sorted copy of the input, and a running time of $\Theta(n\lg n)$ dominated entirely by that sort.

That is a **greedy algorithm**, and this chapter is about it. Chapter 7's closing paragraph promised it, and named the extra ingredient that makes it legal: the *greedy-choice property*. We are now going to define that property, prove it for three different problems, and — crucially — see it fail, so that you never again mistake an attractive rule for a correct one.

9.2 What a Greedy Algorithm Is

9.2.1 The definition

> **DEFINITION.** A **greedy algorithm** solves an optimization problem by making 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 of the problem — never the solutions to the subproblems that the choice will create, and never any future choice. Having made a choice, it never reconsiders it.

Three phrases in that definition carry all the weight, so let us take them one at a time.

"That looks best at that moment." The rule is *local*. It looks at what is in front of it and ranks the available options. In making change it says: take the biggest coin that fits. In the activity-selection problem it will say: take the activity that finishes earliest. In Huffman coding it will say: merge the two least frequent symbols. Each of these is a one-line rule you could give to a shopkeeper.

"Never the solutions to the subproblems." This is the exact point where greedy parts company with dynamic programming. The contrast can be stated plainly: dynamic programming *solves the subproblems before making the first choice*; a greedy algorithm *makes its first choice before solving any subproblem*. A greedy choice may depend on choices already made — after all, they have narrowed the problem — but it may not depend on any future choice or on the value of any subproblem.

"Never reconsiders." There is no backtracking. A greedy algorithm is a one-way street. That is where its speed comes from, and it is also why it can be so badly wrong: a mistake made at step 3 can never be undone at step 40.

> **IN PLAIN LANGUAGE.** You are at a market with a fixed budget and you want to fill your basket as well as possible. A dynamic programmer would work out, for every possible budget and every prefix of the stalls, what the best basket is, and only then start buying. A greedy shopper walks up to the first stall, grabs whatever looks like the best value on the table, pays, and moves on. The greedy shopper is home in time for lunch. Whether the greedy shopper is right depends entirely on what "best value" means and on how the stalls are arranged — and that is the whole subject of this chapter.

9.2.2 The uncomfortable fact

Here is the sentence that most textbooks bury and that this chapter puts first.

> **A GREEDY ALGORITHM IS USUALLY WRONG.** Take a random optimization problem and a plausible local rule, and the algorithm you get will not, in general, produce an optimal solution. It bears saying twice, in two forms: greedy algorithms *do not always yield optimal solutions*,

and the greedy strategy, applied to an arbitrary problem, *does not always produce an optimal solution*. There is nothing in the *form* of a greedy algorithm that makes it correct. Greed is a hope, not a method.

This has a consequence for how you must work, and it is the most important methodological point in the chapter.

THE PROOF IS THE WHOLE JOB. Writing a greedy algorithm is trivial — you sort the input and take things in order. Deciding *which* order, and *proving* that taking things in that order is safe, is the entire task. A greedy algorithm without a correctness proof is not an algorithm; it is a guess with good manners. Conversely, a greedy algorithm *with* a proof is usually the fastest and simplest correct algorithm the problem has.

Compare this with the situation in Chapter 7. A dynamic program that considers every split point k is correct as soon as you have proved optimal substructure and got the recurrence right, because it never gambles: it tries everything. A greedy algorithm gambles at every step. The proof is what converts the gamble into a certainty.

9.2.3 What the rest of the chapter does

The plan is therefore:

1. **§9.3** shows greed failing, on the most everyday problem there is — making change — and then shows the *same rule* succeeding on a different set of coins. This is the lesson in miniature: the rule is not the point, the proof is.
2. **§9.4** works the activity-selection problem completely: the problem, the running example, four candidate greedy rules of which three are wrong, the proof of optimal substructure, and then the **exchange argument** that proves the surviving rule correct. Learn that argument. It reappears in §9.6, in §9.7, in §9.8, and in Chapters 11 and 13.
3. **§9.5** extracts the recipe: the two ingredients a problem needs, and how to test for them.
4. **§9.6** puts greedy and dynamic programming side by side on the two knapsacks, with the arithmetic.
5. **§9.7** builds Huffman codes and proves them optimal.
6. **§9.8** asks the deep question — *is there a structure that characterizes when greed works?* — and answers it with matroids.
7. **§9.9** lists the greedy algorithms you will meet in Chapters 11 and 13, so you recognize them when they arrive.

9.3 A Motivating Failure: Making Change

9.3.1 The problem

You are the cashier at a warung. A customer must be given a certain amount of change, and you want to hand over **as few coins as possible** — your till is small and you would rather not part with fifty one-rupiah pieces.

DEFINITION (the coin-change problem). You are given a set of coin denominations $d_1, d_2, \ldots, d_k$, all positive integers, and an amount A, also a positive integer. You have an unlimited supply of each denomination. Find a multiset of coins whose values sum to exactly A and whose **number of coins** is as small as possible.

Two symbols there, both new, so let us be explicit. A *denomination* is the face value of one kind of coin — in Indonesia, as of September 2026, Bank Indonesia issues banknotes of 1000, 2000, 5000,

10 000, 20 000, 50 000 and 100 000 rupiah (the 2022 series) and coins of 100, 200, 500 and 1000 rupiah [1]. Leaving aside the three small coins, which rarely change hands, the denominations we pay with are 1000, 2000, 5000, 10 000, 20 000, 50 000 and 100 000 rupiah, so $k = 7$ and $d_1 = 1000$, $d_2 = 2000$, and so on. A *multiset* is like a set except that repeats are allowed and counted: {5000,5000,1000} is a legitimate answer with three coins. The amount A is what we owe.

9.3.2 The greedy rule

Everyone invents the same rule, and it is the right instinct:

> **THE GREEDY CHANGE RULE.** While money is still owed, hand over the **largest coin that does not exceed the amount still owed**. Repeat.

It is local: it looks only at the amount still owed. It never reconsiders: once a coin is out of the till it stays out. It is obviously fast. And it feels right, because a big coin does more work than a small one.

Work it by hand for $A = 78\,000$ rupiah with the Indonesian denominations.

- Owed 78 000. Largest coin that fits: 50 000. Hand it over. Owed $78\,000 - 50\,000 = 28\,000$.
- Owed 28 000. Largest coin that fits: 20 000. Hand it over. Owed $28\,000 - 20\,000 = 8\,000$.
- Owed 8 000. Largest coin that fits: 5 000. Hand it over. Owed $8\,000 - 5\,000 = 3\,000$.
- Owed 3 000. Largest coin that fits: 2 000. Hand it over. Owed $3\,000 - 2\,000 = 1\,000$.
- Owed 1 000. Largest coin that fits: 1 000. Hand it over. Owed 0. Stop.

Five coins: $50\,000 + 20\,000 + 5\,000 + 2\,000 + 1\,000 = 78\,000$. And indeed no four coins can do it, as we will confirm by machine in §9.3.5.

9.3.3 The counterexample

Now change the coins. Suppose — and this is not fanciful, it is roughly the pre-decimal British situation and exactly the situation in several board games — the denominations are

$$d_1 = 1, \qquad d_2 = 3, \qquad d_3 = 4,$$

and the amount owed is $A = 6$.

Run the greedy rule, one step per line, with the arithmetic shown.

- Owed 6. Largest coin that does not exceed 6: that is 4, since $4 \leq 6$ but the next denomination up does not exist. Hand over 4. Owed $6 - 4 = 2$.
- Owed 2. Largest coin that does not exceed 2: not 4 (too big), not 3 (too big), so 1. Hand over 1. Owed $2 - 1 = 1$.
- Owed 1. Largest coin that fits: 1. Hand over 1. Owed $1 - 1 = 0$. Stop.

Greedy has handed over $4 + 1 + 1$, which is **three coins**.

Now look at what a thoughtful cashier does:

$$3 + 3 = 6.$$

Two coins. Greedy used three where two suffice — fifty percent more coins than necessary, on an amount of six, with three denominations. This is not a pathological hundred-denomination construction; it is the smallest interesting instance there is.

WHY GREED FAILED HERE. The greedy rule took the 4 because $4 > 3$, and a bigger coin *seems* to make more progress. But taking the 4 left a remainder of 2, and 2 is a terrible amount to owe in this system: the only way to pay it is two 1s. Taking the smaller coin 3 left a remainder of 3, which is payable with a single coin. The greedy rule optimized the wrong thing. It maximized *this step's* progress and thereby ruined *the next step's* options — and, having no backtracking, it could not recover.

That is the general anatomy of greedy failure, and it is worth naming.

THE SHAPE OF EVERY GREEDY FAILURE. A greedy rule fails when the locally attractive choice damages the subproblem it leaves behind, by more than the local gain was worth. The rule cannot see that damage, because by definition it does not look at the subproblem.

A second, even more famous counterexample: denominations {1,10,25} — American coins with the five-cent piece removed — and $A = 30$. Greedy takes 25, leaving 5, which must be paid as $1 + 1 + 1 + 1 + 1$: six coins in total. The optimum is $10 + 10 + 10$: three coins. Greedy used **twice** as many coins as necessary.

9.3.4 A coin system where greed succeeds

Now the other half of the lesson. Put the five-cent piece back: denominations {1,5,10,25}, the actual American coin set. Try to break greedy and you will not manage it — for *every* amount, the greedy rule produces a minimum-size multiset. The same is true of the Indonesian set {1,2,5,10,20,50,100} in thousands of rupiah, and of the powers of two {1,2,4,8,16,32,64}.

DEFINITION. A set of denominations is called **canonical** if the greedy change rule returns a minimum-size multiset for every amount. {1,5,10,25} is canonical. {1,3,4} is not.

So the identical rule — "biggest coin that fits" — is *provably optimal* on one coin set and *provably wrong* on another. Nothing about the rule changed. What changed was the structure of the input, and therefore whether a proof exists.

THE LESSON OF THIS SECTION, IN ONE SENTENCE. The strategy is not the point; the proof is the point. "Be greedy" is not an algorithm design technique on its own — it becomes one only when paired with an argument that this particular greed, on this particular problem, cannot go wrong.

It is worth adding that deciding whether an arbitrary coin system is canonical is itself a real problem, not a triviality. There is no simple pattern to read off the denominations. (Pearson's algorithm decides it in $O(k^3)$ time for k denominations, by checking a bounded set of candidate counterexamples [2]; and note that the general coin-change problem, for non-canonical systems, is exactly the kind of thing Chapter 7's dynamic programming was built for.)

9.3.5 Measured

Listing 9.1 audits six coin systems. For each system and for every amount from 1 to 200 it computes the greedy answer and the true optimum — the latter by the dynamic program of Chapter 7, a one-dimensional table with `best[a] = 1 + min over coins c of best[a-c]` — and reports the smallest amount, if any, at which the two disagree.

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;
}
```

The program printed:

```
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
```

Read the numbers carefully, because they are more alarming than the hand counterexamples suggested. On the system {1,3,4}, greedy is not wrong at one unlucky amount — it is wrong at **49 of the first 200 amounts**, roughly one in four. On {1,10,25} it is wrong at **70 of 200**, more than one in three. A greedy rule that fails does not usually fail rarely; it fails systematically, and the failures are invisible unless you test against a known-correct method. **Table 9.1** collects the result.

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
{1,2,5,10,20,50,100}	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

Notice the last row. Dropping the 2000-rupiah note from the Indonesian set leaves {1,5,10,20,50,100}, which is *still* canonical — so removing a denomination need not break greed. And notice, by contrast, that {1,10,25} differs from the canonical {1,5,10,25} by a single denomination, and that one omission breaks it at 70 amounts out of 200. There is no shortcut. You must prove it.

> **A FAMILIAR ECHO.** Exercise 7.18 asked you to pack a Profitina Laundry order into bags of 3 kg, 5 kg and 8 kg, filled exactly, using as few bags as possible, and to show that "use the biggest bag that fits" fails. It does, and now you know the name of the phenomenon: {3,5,8} is not canonical. For a 9 kg order, greedy takes the 8 kg bag and is left with 1 kg it cannot bag at all; three 3 kg bags do the job.

9.4 The Activity-Selection Problem

This is the problem on which the whole method is built, and we are going to do it completely: the statement, the data, four candidate rules of which three are wrong, optimal substructure proved, the greedy-choice property proved by an exchange argument, the recursive algorithm, the iterative algorithm, the trace, and the running time.

9.4.1 The problem

DEFINITION (the activity-selection problem). We are given a set $S = \{a_1, a_2, \dots, a_n\}$ of n proposed **activities**, all of which want exclusive use of a single common resource — one meeting room, one washing machine, one telescope. Activity a_i has a **start time** s_i and a **finish time** f_i with $0 \le s_i < f_i < \infty$. If a_i is selected, it occupies the resource during the **half-open interval** $[s_i, f_i)$. Two activities a_i and a_j are **compatible** if their intervals do not overlap, which happens exactly when

$$s_i \ge f_j \qquad \text{or} \qquad s_j \ge f_i.$$

The goal is to select a **maximum-size** subset of mutually compatible activities.

Every symbol in that definition needs a word of explanation, because the details matter.

S is just the name of the whole collection of requests. n is how many there are. s_i and f_i are two numbers attached to request number i — the clock time at which it wants to start and the clock time at which it will be finished. They can be minutes after opening, or hours, or any unit; all that matters is that we can compare them. The condition $s_i < f_i$ says an activity takes a positive amount of time, and $f_i < \infty$ says it does eventually end.

Why half-open? The notation $[s_i, f_i)$ means "from s_i inclusive up to f_i exclusive". The square bracket includes the endpoint; the round bracket excludes it. This is exactly the right convention for scheduling, because it makes an activity finishing at 10:00 and another starting at 10:00 compatible — the resource is handed over at the instant 10:00 and no time is double-booked. If we had used the closed interval $[s_i, f_i]$, those two activities would technically overlap at the single instant 10:00, which is not how a laundry or a meeting room works.

Why that compatibility test? The two clauses cover the only two ways two intervals can miss each other: either a_i starts at or after a_j has finished ($s_i \ge f_j$, so a_j comes first), or a_j starts at or after a_i has finished ($s_j \ge f_i$, so a_i comes first). If neither holds, each starts before the other finishes, and they overlap.

Note what is *not* being maximized. We are maximizing the **number** of activities, not the total time the resource is busy, and not any notion of value or revenue. This matters enormously: §9.14 (Exercise 9.15) shows that as soon as activities carry different values, the greedy rule we are about to prove correct becomes wrong, and the problem needs dynamic programming again.

Finally, we adopt a standing assumption that will pay for itself many times over:

$$f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n.$$

(9.1)

In words, Equation (9.1) says that the activities are given **sorted into monotonically increasing order of finish time**. If they are not, sort them; that costs $O(n\lg n)$ and, as we shall see in §9.4.11, it turns out to be the dominant cost of the whole algorithm.

9.4.2 The running example

Figure 9.1 is the instance we shall use throughout this section.

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 $f_1 \leq f_2 \leq \cdots \leq f_{11}$. This one table is the running example for the whole of §9.4.

Read the table as three rows: the activity number i, its start time s_i, and its finish time f_i. So a_1 runs during [1,4), a_2 during [3,5), a_3 during [0,6), and so on up to a_{11} during [12,16). Check the sort required by Equation (9.1): the finish times read 4,5,6,7,9,9,10,11,12,14,16 — monotonically increasing (with a tie at 9, which is permitted, since the order is *non-decreasing*).

Table 9.2 restates the same data with the duration computed, because we will want it in §9.4.4.

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

Two facts about this instance deserve attention, and both are instructive.

First: the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. Check it. $a_3 = [0,6)$ and $a_9 = [8,12)$: is $s_9 \geq f_3$? Yes, $8 \geq 6$. Compatible. $a_9 = [8,12)$ and $a_{11} = [12,16)$: is $s_{11} \geq f_9$? Yes, $12 \geq 12$ — and here the half-open convention earns its keep. Compatible. a_3 and a_{11}: $12 \geq 6$. Compatible. So all three pairs are compatible and the set is legal. But it is **not** a maximum set: it has only three activities.

Second: the subsets $\{a_1, a_4, a_8, a_{11}\}$ and $\{a_2, a_4, a_9, a_{11}\}$ each have four activities, and four is the most possible. Check the first: $a_1 = [1,4)$, $a_4 = [5,7)$ — $5 \geq 4$ ✓; a_4, $a_8 = [8,11)$ — $8 \geq 7$ ✓; a_8, $a_{11} = [12,16)$ — $12 \geq 11$ ✓. And since compatibility here is just "each starts after the previous one finished", the non-adjacent pairs are compatible automatically.

> **MAXIMAL IS NOT MAXIMUM.** A set is **maximal** if you cannot add any further activity to it. A set is **maximum** if no other legal set is larger. $\{a_3, a_9, a_{11}\}$ is maximal — try adding anything and it clashes — but it is not maximum, because $\{a_1, a_4, a_8, a_{11}\}$ is bigger. Maximal is a local property you can check by inspection; maximum is a global property that requires an argument. Every greedy algorithm produces a maximal solution effortlessly; the entire difficulty is in getting a maximum one.

Note too that the maximum set is **not unique**: $\{a_1, a_4, a_8, a_{11}\}$ and $\{a_2, a_4, a_9, a_{11}\}$ are both optimal. This is normal, and it is important for the proofs ahead: the greedy-choice property will say that *some* optimal solution contains the greedy choice, not that *every* one does.

9.4.3 Four candidate greedy rules

Before proving anything, let us be honest about how many plausible rules there are. Each of these is a one-line local rule; each has been proposed by students; three of them are wrong.

1. **Earliest start time.** Take the activity that starts soonest, discard everything that clashes with it, repeat. *Reasoning:* get going as early as possible.
2. **Shortest duration.** Take the activity that occupies the resource for the least time, discard clashes, repeat. *Reasoning:* short activities leave more room for others.
3. **Fewest conflicts.** Count, for each activity, how many other remaining activities clash with it; take the one with the fewest, discard clashes, repeat. *Reasoning:* the least disruptive activity should be safest.
4. **Earliest finish time.** Take the activity that finishes soonest, discard clashes, repeat. *Reasoning:* free the resource again as early as possible.

Run all four on the running example and something awkward happens: two of the wrong ones get the right answer. From the verified output of **Listing 9.2**:

```
=== 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
```

Earliest-start already fails on this very instance: it grabs $a_3 = [0,6)$, which blocks a_1, a_2, a_4, a_5, a_6 and a_{10}, then $a_7 = [6,10)$, then a_{11}, for a total of three instead of four. That is exactly the shape of failure named in §9.3.3: a_3 starts earliest but hogs six time units.

Shortest-duration and fewest-conflicts, however, get four on this instance. **This is the trap.** One instance proves nothing. To show a rule is wrong you need a counterexample; to show it is right you need a proof. **Listing 9.2** searches exhaustively for the smallest counterexample to each, and found these:

```
=== 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
```

Walk through the shortest-duration counterexample by hand, because it is only three activities and it is completely convincing. The activities are $a_1 = [5,8)$ of duration 3, $a_2 = [1,5)$ of duration 4, and $a_3 = [4,6)$ of duration 2. The shortest is a_3, duration 2. Take it. Now $a_1 = [5,8)$: is $5 \geq 6$? No. Is $8 \leq 4$? No. They overlap on $[5,6)$, so a_1 is discarded. And $a_2 = [1,5)$: is $4 \geq 5$? No. Is $5 \leq 1$? No. They overlap on $[4,5)$, so a_2 is discarded too. The rule ends with **one** activity. But $a_2 = [1,5)$ and $a_1 = [5,8)$ are compatible ($5 \geq 5$ ✓), so **two** was available. The short activity sat in the middle and poisoned both of its neighbors.

The fewest-conflicts counterexample needs nine activities — no smaller instance breaks it, which is exactly why the rule is so seductive — and its trace is worked out in full in §9.14, Exercise 9.4.

WHY EARLIEST FINISH IS DIFFERENT. Here is the intuition, and it is the seed of the proof. *We should choose an activity that leaves the resource available for as many other activities as possible.* Of all the activities we end up choosing, one of them must be the first to finish. The activity in S with the **earliest finish time** leaves the resource free from the earliest possible moment onwards — no other choice can free it sooner. Since the activities are sorted by finish time, the greedy choice is simply a_1.

That is an argument about *intuition*, not a proof. We now make it a proof. There are two things to establish, and they correspond exactly to the two ingredients that every greedy proof needs: **optimal substructure**, and the **greedy-choice property**.

9.4.4 Optimal substructure, proved

First we must see the problem as a collection of subproblems. Define, for two indices i and j,

$$S_{ij} = \{ a_k \in S : f_i \le s_k \text{ and } f_k \le s_j \}, \tag{9.2}$$

in words: S_{ij} **is the set of activities that start after activity** a_i **finishes and that finish before activity** a_j **starts.** It is the set of activities that could be squeezed into the gap between a_i and a_j.

To make the definition Equation (9.2) meaningful for the whole problem we add two fictitious activities: a_0, with $f_0 = 0$, and a_{n+1}, with $s_{n+1} = \infty$. Then $S_{0,n+1}$ is the entire set S, because every real activity starts after time 0 and finishes before time ∞. With this device, "solve the original problem" and "solve the subproblem S_{ij}" become the same kind of sentence, which is what we need for a recursion.

Now the theorem.

THEOREM 9.1 (optimal substructure of activity selection). Let A_{ij} be a maximum-size set of mutually compatible activities in S_{ij}, and suppose $a_k \in A_{ij}$ for some activity a_k. Define

$$A_{ik} = A_{ij} \cap S_{ik}, \qquad A_{kj} = A_{ij} \cap S_{kj}.$$

Then $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, so that

$$|A_{ij}| = |A_{ik}| + |A_{kj}| + 1, \tag{9.3}$$

and moreover A_{ik} is a maximum-size compatible set in S_{ik} and A_{kj} is a maximum-size compatible set in S_{kj}.

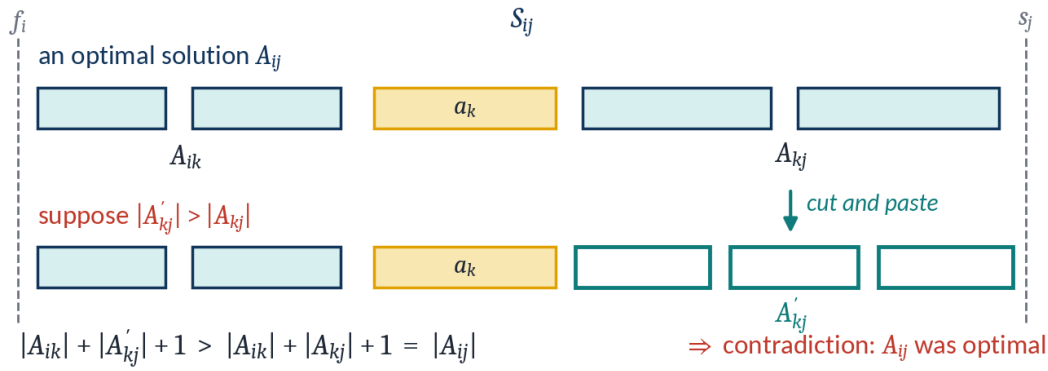
Proof. We prove the three claims in order.

Step 1: the decomposition covers everything. Take any activity $a_m \in A_{ij}$ other than a_k. Since A_{ij} is a set of mutually compatible activities and both a_m and a_k are in it, a_m and a_k are compatible. So either $f_m \le s_k$ or $f_k \le s_m$.

Step 2: each element lands in exactly one piece. Suppose $f_m \le s_k$, that is, a_m finishes before a_k starts. We also know $a_m \in S_{ij}$, so $f_i \le s_m$. Together, $f_i \le s_m$ and $f_m \le s_k$, which is precisely the condition for $a_m \in S_{ik}$. Hence $a_m \in A_{ij} \cap S_{ik} = A_{ik}$. Symmetrically, if $f_k \le s_m$ then $a_m \in S_{kj}$ and so $a_m \in A_{kj}$. The two cases cannot both hold, since $s_k < f_k$ would force $f_m \le s_k < f_k \le s_m < f_m$,

an impossibility. So every element of A_{ij} other than a_k is in exactly one of A_{ik}, A_{kj}, giving $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ as a disjoint union, and therefore the count Equation (9.3) holds. ■ (first two claims)

Step 3: the pieces are optimal — the cut-and-paste argument. This part is a proof by contradiction, and **Figure 9.2** states it in full.



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}.

Here is the argument written out one step per line.

1. Suppose, for contradiction, that A_{kj} is *not* a maximum-size compatible set in S_{kj}.
2. Then there exists a set A'_{kj} of mutually compatible activities in S_{kj} with $|A'_{kj}| > |A_{kj}|$.
3. Form the set $B = A_{ik} \cup \{a_k\} \cup A'_{kj}$. This is the **cut and paste**: we cut A_{kj} out of A_{ij} and paste A'_{kj} in.
4. **Is B legal?** This is the step §7.5.3 warned you never to skip. Every activity in A_{ik} finishes at or before s_k; every activity in A'_{kj} starts at or after f_k; and $s_k < f_k$. So every activity of A_{ik} finishes before every activity of A'_{kj} starts, and a_k sits between them without touching either. Activities within A_{ik} are mutually compatible because $A_{ik} \subseteq A_{ij}$; activities within A'_{kj} are mutually compatible by assumption. Therefore all pairs in B are compatible: B is legal.
5. **How big is B?** The union in step 3 is disjoint for the reasons just given, so $|B| = |A_{ik}| + |A'_{kj}| + 1$.
6. Using $|A'_{kj}| > |A_{kj}|$,

$$|B| = |A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|.$$

7. But B is a set of mutually compatible activities all lying in S_{ij} — it is a legal solution to the very subproblem A_{ij} was supposed to solve optimally — and it is strictly bigger. That contradicts the assumption that A_{ij} is maximum-size. 8. Hence no such A'_{kj} exists, and A_{kj} is maximum-size after all. A symmetric argument, exchanging the roles of the two sides, shows the same for A_{ik}. ■

That is a complete proof of optimal substructure, and it uses exactly the machinery of §7.5.3: assume a piece is suboptimal, substitute a better piece, verify the result is legal, derive a contradiction.

9.4.5 The dynamic program we are about to throw away

Theorem 9.1 immediately gives a dynamic program, whose recurrence already appears at the foot of **Figure 9.2**. Let $c[i, j]$ be the size of a maximum-size compatible set in S_{ij}. If S_{ij} is empty there is nothing to choose and $c[i, j] = 0$. Otherwise some activity $a_k \in S_{ij}$ is in the optimal set, and by Theorem 9.1 and Equation (9.3) the total is $c[i, k] + c[k, j] + 1$ — but we do not know which a_k, so we try them all. The result is the recurrence Equation (9.4):

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max\limits_{a_k \in S_{ij}} \{\, c[i, k] + c[k, j] + 1 \,\} & \text{if } S_{ij} \neq \emptyset. \end{cases} \tag{9.4}$$

This is a perfectly good algorithm. It is also, structurally, the matrix-chain algorithm of §7.6 all over again: a two-dimensional table with a maximization over an inner index, hence $\Theta(n^3)$ work in the worst case. You could memoize it (§7.7) or fill it bottom-up (§7.4.2); either way you would pay $\Theta(n^2)$ space and roughly $n^3/6$ operations.

We are going to do far better, because for *this* problem the maximization is unnecessary. The whole point of the next section is that we can identify the winning a_k by inspection.

9.4.6 The greedy choice, and why only one subproblem remains

Figure 9.3 summarizes this step.

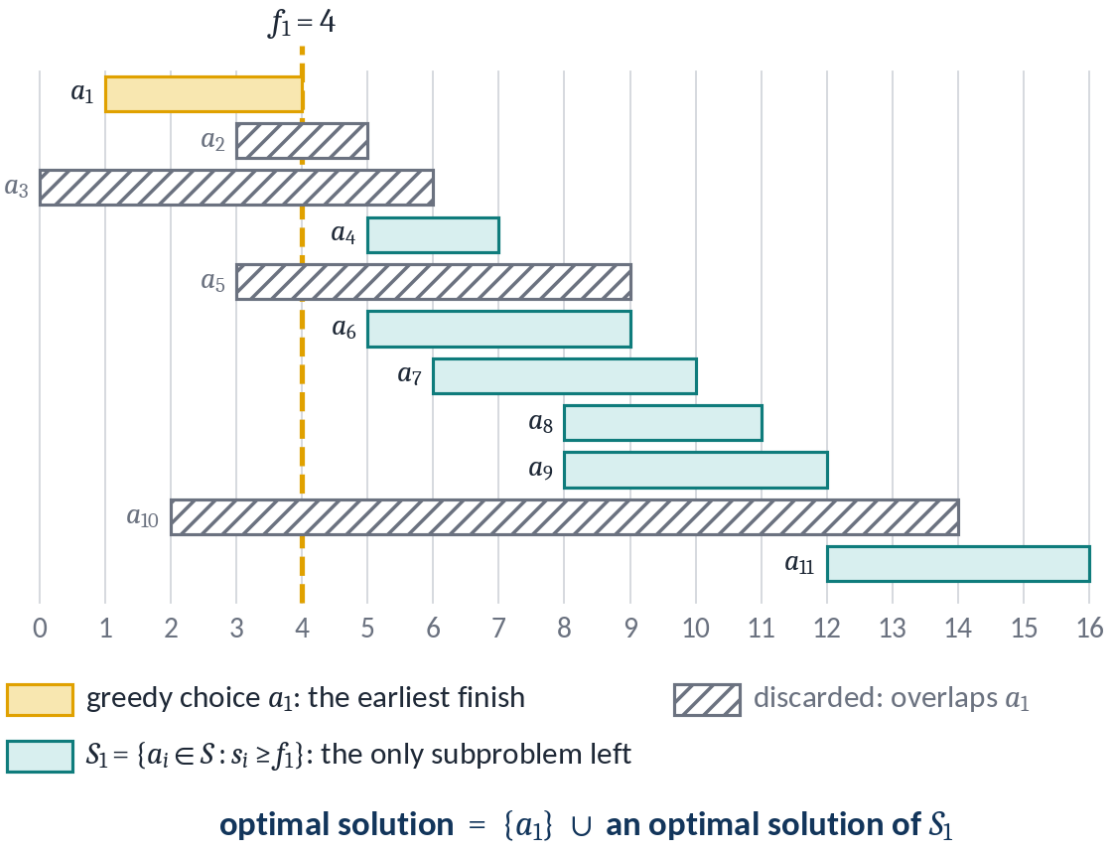


Figure 9.3 The reduction to a single subproblem: once the greedy choice a_1 has been made, the single subproblem S_1 remains, and an optimal solution to the original problem is a_1 together with an optimal solution to S_1.

Two things are happening in that figure, and they are worth separating.

First, a simplification of the subproblem space. Instead of the two-index family S_{ij}, we define the one-index family

$$S_k = \{\, a_i \in S \,:\, s_i \geq f_k \,\}, \tag{9.5}$$

the set of activities that start after activity a_k finishes. Why is one index enough? Because of a fact about the *right-hand* end. If we always choose the earliest-finishing available activity, then having chosen a_k we never need to bound the far end of the gap: every remaining activity we could possibly use starts after f_k, and nothing is excluded from the right. Formally, the subproblems that arise from a greedy run are exactly $S_1 \subset S_{m_1} \subset \cdots$ — a chain, not a table.

Second, the reduction to one subproblem. The argument is short and complete, and we spell it out one step per line.

1. Suppose we choose a_1, the activity with the earliest finish time. (The sort guarantees $f_1 \leq f_i$ for all i.)
2. What remains to solve is: find a maximum-size compatible set among the activities compatible with a_1.
3. **Claim:** every activity compatible with a_1 starts at or after f_1 — that is, the remaining problem is exactly S_1 as defined by Equation (9.5), with nothing "before a_1" to worry about.
4. *Proof of the claim.* Let a_i be compatible with a_1. By the definition of compatibility, either $s_i \geq f_1$ or $s_1 \geq f_i$. Suppose the second held: $s_1 \geq f_i$. Since $s_1 < f_1$ we would get $f_i \leq s_1 < f_1$, so $f_i < f_1$

— an activity finishing strictly before a_1. But f_1 is the *minimum* finish time over all of S, so no such activity exists. Hence the second case is impossible and $s_i \geq f_1$. ■

5. Therefore after choosing a_1 there is exactly **one** subproblem: S_1. Not two, as in Theorem 9.1; not n candidates to try, as in the recurrence of §9.4.5. One.

> **IN PLAIN LANGUAGE.** Because a_1 is the very first activity in the whole set to finish, nothing can sneak in *before* it — anything that ended earlier would contradict a_1 being first to finish. So the only region left to fill is the time after f_1, and that is a single smaller instance of exactly the same problem.

9.4.7 The greedy-choice property, proved by an exchange argument

We now have the shape of a greedy algorithm, but not yet the right to use it. What is missing is the guarantee that taking a_1 does not cost us anything — that we are not in the {1,3,4} coin situation, grabbing the big attractive coin and ruining what follows.

> **THEOREM 9.2 (the greedy-choice property for activity selection).** Consider any non-empty subproblem S_k, and let a_m be an activity in S_k with the **earliest finish time**, that is, $f_m = \min\{f_i : a_i \in S_k\}$. Then a_m is included in **some** maximum-size subset of mutually compatible activities of S_k.

Read the statement carefully. It does not say a_m is in *every* optimal solution — §9.4.2 already showed optimal solutions need not be unique. It says there is *at least one* optimal solution containing a_m. That is exactly enough: it means choosing a_m cannot lock us out of optimality.

Proof (the exchange argument). We take an arbitrary optimal solution and *transform* it into one that contains a_m, without making it any smaller. Because we can always do that, an optimal solution containing a_m must exist.

1. Let A_k be **any** maximum-size subset of mutually compatible activities of S_k. (One exists because S_k is a finite non-empty set, so it has at least one legal subset, and among the finitely many legal subsets some is largest.)
2. Order the activities of A_k by finish time, and let a_j be the **first** one to finish, that is, the element of A_k with the smallest finish time.
3. **Case 1:** $a_j = a_m$**.** Then A_k itself is a maximum-size solution containing a_m, and we are finished — there is nothing to prove.
4. **Case 2:** $a_j \neq a_m$**.** Build a new set by exchanging a_j for a_m:

$$A'_k = (A_k \setminus \{a_j\}) \cup \{a_m\}.$$

- This is the **exchange**; everything now turns on two questions about A'_k — is it legal, and how big is it?

5. **How big is** A'_k**?** We removed exactly one element and added exactly one element. We must check the added element was not already present: $a_m \neq a_j$ by the assumption of Case 2, and $a_m \notin A_k \setminus \{a_j\}$ — because if a_m were some other element of A_k, then f_m would be the finish time of an element of A_k larger than the minimum f_j, giving $f_m > f_j$; but f_m is the minimum finish time over all of $S_k \supseteq A_k$, so $f_m \leq f_j$, a contradiction. Hence

$$|A'_k| = |A_k| - 1 + 1 = |A_k|.$$

- The exchange is **size-preserving**. This is the heart of the technique: we do not claim the greedy choice is *better*, only that it is *no worse*.

6. **Is A'_k legal?** We must show every pair in A'_k is compatible.
   - Pairs not involving a_m: both elements come from A_k, which was mutually compatible. ✓
   - Pairs involving a_m: let a_i be any element of $A_k \setminus \{a_j\}$. We must show a_m and a_i are compatible, and we show the stronger statement $f_m \leq s_i$.
     - Since a_j is the first activity of A_k to finish, and $a_i \in A_k$ with $a_i \neq a_j$, we have $f_j \leq f_i$, and since a_i and a_j are compatible (both in A_k) with a_j finishing no later, the only possibility is $f_j \leq s_i$. (The alternative, $f_i \leq s_j$, would give $f_i \leq s_j < f_j \leq f_i$, which is impossible.)
     - Since a_m has the minimum finish time in the whole of S_k, and $a_j \in A_k \subseteq S_k$, we have $f_m \leq f_j$.
     - Chaining the two: $f_m \leq f_j \leq s_i$, so $f_m \leq s_i$, which is the compatibility condition $s_i \geq f_m$. ✓
   - Every pair is compatible, so A'_k is a legal set of mutually compatible activities, and every element of it lies in S_k.
7. So A'_k is a legal subset of S_k of mutually compatible activities with $|A'_k| = |A_k|$, which is the maximum size. Therefore A'_k is itself a maximum-size solution, **and it contains** a_m.
8. In both cases a maximum-size solution containing a_m exists. ■

> **THE EXCHANGE ARGUMENT, AS A TEMPLATE.** Every greedy correctness proof in this book has the shape you have just read. Memorize the four moves. **(1) Take an arbitrary optimal solution** A — not a convenient one, any one. **(2) Locate the element of** A **that plays the role your greedy choice wants to play.** Here: the first activity of A to finish. **(3) Swap your greedy choice in for it**, and prove two things: the result is still **feasible**, and it is **no smaller** (or no more expensive). **(4) Conclude that an optimal solution containing the greedy choice exists**, hence the greedy choice is *safe*. Then, separately, prove optimal substructure so the argument can be repeated on what remains. The two together, by induction on the number of choices, prove the whole algorithm optimal.

9.4.8 Putting the two halves together

Theorems 9.1 and 9.2 combine into the statement the algorithm actually needs.

> **COROLLARY 9.3.** Let S_k be a non-empty subproblem and let $a_m \in S_k$ have the earliest finish time in S_k. Let A^* be a maximum-size compatible subset of S_m. Then $\{a_m\} \cup A^*$ is a maximum-size compatible subset of S_k.

Proof. By Theorem 9.2 there is a maximum-size compatible subset B of S_k with $a_m \in B$. By the argument of §9.4.6 step 4, applied inside S_k, every activity of B other than a_m starts at or after f_m, so $B \setminus \{a_m\}$ is a compatible subset of S_m, giving $|B| - 1 \leq |A^*|$. Hence $|B| \leq |A^*| + 1 = |\{a_m\} \cup A^*|$. Conversely $\{a_m\} \cup A^*$ is compatible (every element of $A^* \subseteq S_m$ starts at or after f_m) and lies in S_k, so it is a legal solution and $|\{a_m\} \cup A^*| \leq |B|$ by the maximality of B. The two inequalities give equality, so $\{a_m\} \cup A^*$ has maximum size. ■

Corollary 9.3 is the license to recurse. It says: take the earliest finisher, then solve *one* smaller problem of the same kind, and glue. By induction on the number of activities remaining — the base case being the empty subproblem, which has the empty set as its unique and therefore optimal solution — the algorithm of the next section is correct.

9.4.9 The recursive greedy algorithm

Figure 9.4 gives the pseudocode.

```
    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 $f_0 = 0$.

Line by line, in words:

- **Line 1**, `m = k + 1`. Start looking at the activity just after a_k. Because the array is sorted by finish time, scanning forward from $k + 1$ visits activities in increasing order of finish time, so the *first* suitable one we meet is the earliest finisher.
- **Line 2**, `while m ≤ n and s[m] < f[k]`. Keep advancing while activity a_m is *not* in S_k. Recall S_k requires $s_m \geq f_k$; so $s_m < f_k$ means a_m starts too early and must be skipped.
- **Line 3**, `m = m + 1`. Advance.
- **Line 4**, `if m ≤ n`. Did we find one before running off the end?
- **Line 5**, `return {a_m} ∪ Recursive-Activity-Selector(s, f, m, n)`. Take a_m — this is the greedy choice, justified by Theorem 9.2 — and recurse on the one remaining subproblem S_m, justified by Corollary 9.3.
- **Line 6**, `else return ∅`. S_k was empty; the empty set is its optimal solution.

The initial call is `Recursive-Activity-Selector(s, f, 0, n)`. The sentinel a_0 has $f_0 = 0$, so the test `s[m] < f[0]` is `s[m] < 0`, which is false for every real activity — the loop does not skip anything and m ends at 1, correctly making a_1 the first greedy choice.

> **IT IS NOT REALLY RECURSION.** Notice that the recursive call in line 5 is the **last** thing the procedure does with its own work — it is a *tail call*. Nothing is computed after the call returns except gluing a_m onto the front of the answer. A recursion of that shape is a loop in disguise, which is exactly why §9.4.11 can rewrite it as one. Contrast Chapter 7's matrix-chain recursion, which makes two calls and then combines their results; that one is genuinely recursive and cannot be flattened into a single loop.

9.4.10 The trace on the running example

Figure 9.5 is the complete trace of **Recursive-Activity-Selector** on the eleven activities of **Figure 9.1**. It is the single most informative picture in the chapter; study it before reading on.

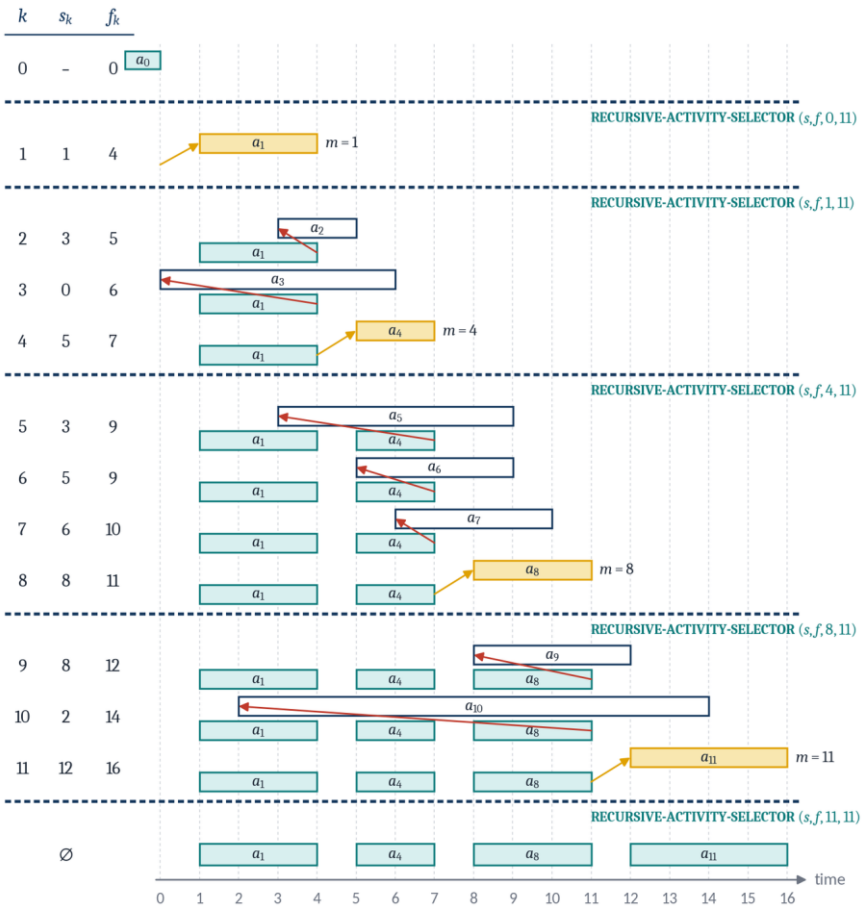


Figure 9.5 The trace of **Recursive-Activity-Selector** on the eleven activities of **Figure 9.1**. Each row is one iteration of the **while** loop: the upper bar is activity a_k (amber when it is taken), the teal bars are activities already selected, and the arrow marks the comparison s_k against f 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.

Here is how to read it. The leftmost three columns repeat k, s_k and f_k from the data table. Each row of the diagram corresponds to one value of m examined by the **while** loop. The upper bar in each row is the activity being examined — white when it is skipped, amber when it is taken; the teal bars below it are the activities already chosen. A red arrow pointing left from the examined bar back to the last teal bar means "this activity starts before that one finishes, so skip it"; an amber arrow pointing right means the activity starts after the last selection finishes. A row labeled with `m = …` is a row where the loop stopped and the activity was taken.

Now the trace in words, with every comparison shown.

Call 1: `Recursive-Activity-Selector(s, f, 0, 11)`. m starts at 1. Test `s[1] < f[0]`, that is $1 < 0$? **No.** The loop body never runs. $m = 1 \leq 11$, so take a_1 and recurse with $k = 1$.

Call 2: `Recursive-Activity-Selector(s, f, 1, 11)`. Now $f_k = f_1 = 4$. * $m = 2$: is $s_2 < 4$? $3 < 4$ — **yes**, skip. (a_2 starts at 3, while a_1 is still running.) * $m = 3$: is $s_3 < 4$? $0 < 4$ — **yes**, skip. * $m = 4$: is $s_4 < 4$? $5 < 4$ — **no**. Stop. $m = 4$, take a_4, recurse with $k = 4$.

Call 3: `Recursive-Activity-Selector(s, f, 4, 11)`. Now $f_k = f_4 = 7$. * $m = 5$: $s_5 = 3 < 7$ — skip. * $m = 6$: $s_6 = 5 < 7$ — skip. * $m = 7$: $s_7 = 6 < 7$ — skip. * $m = 8$: $s_8 = 8 < 7$? **No**. Stop. Take a_8, recurse with $k = 8$.

Call 4: `Recursive-Activity-Selector(s, f, 8, 11)`. Now $f_k = f_8 = 11$. * $m = 9$: $s_9 = 8 < 11$ — skip. * $m = 10$: $s_{10} = 2 < 11$ — skip. * $m = 11$: $s_{11} = 12 < 11$? **No**. Stop. Take a_{11}, recurse with $k = 11$.

Call 5: `Recursive-Activity-Selector(s, f, 11, 11)`. m starts at 12. The loop condition requires $m \leq n = 11$, which fails immediately. Then `if m ≤ n` is false, so return ∅.

Unwinding: $\{a_{11}\} \cup \varnothing$, then $\{a_8, a_{11}\}$, then $\{a_4, a_8, a_{11}\}$, then $\{a_1, a_4, a_8, a_{11}\}$. Four activities, and §9.4.2 already told us four is the maximum. The machine agrees, from the verified output of **Listing 9.2**:

```
=== 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
```

The iterative form and the exhaustive check follow it in the same run, and they agree:

```
=== GREEDY-ACTIVITY-SELECTOR(s, f) ===
  result = {a1, a4, a8, a11}, size 4
  same answer as the recursive form: yes

=== Brute force over all 2^11 = 2048 subsets ===
  true maximum size = 4
```

> **COUNT THE WORK.** Across the five calls, the variable m took the values 1; 2,3,4; 5,6,7,8; 9,10,11; and 12. Every index from 1 to 11 was examined **exactly once**, plus one final test that ran off the end. That is the key observation for the running time: although the procedure is recursive, the total number of loop iterations over all calls is n, because m never moves backward.

9.4.11 The iterative greedy algorithm

The last step in developing a greedy solution is to convert the recursion into a loop. **Figure 9.6** is the result.

```
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.

Line by line:

- **Line 1**, `n = s.length`. How many activities there are.
- **Line 2**, `A = {a1}`. Take a_1 unconditionally. This is legal because the array is sorted by finish time, so a_1 is the earliest finisher in all of S, and Theorem 9.2 says it is safe.
- **Line 3**, `k = 1`. Remember that a_1 was the last thing selected.
- **Line 4**, `for m = 2 to n`. Walk forward through the remaining activities in increasing order of finish time.
- **Line 5**, `if s[m] ≥ f[k]`. Is a_m compatible with the last selection? Because the selections so far are ordered and non-overlapping, compatibility with the *last* one implies compatibility with all of them.
- **Line 6**, `A = A ∪ {a_m}`. Select it.
- **Line 7**, `k = m`. It is now the last selection.
- **Line 8**, `return A`.

The **loop invariant** — the technique of §2.3 — makes the correctness precise.

> **LOOP INVARIANT.** At the start of each iteration of the **for** loop of line 4, A is a maximum-size subset of mutually compatible activities of $\{a_1, \ldots, a_{m-1}\}$, and a_k is the activity in A with the latest finish time. **Initialization.** Before the first iteration, $m = 2$, so the set is $\{a_1\}$; a maximum-size compatible subset of a one-element set is that element, and $k = 1$. ✓ **Maintenance.** Consider iteration m. If $s_m < f_k$ then a_m overlaps a_k; since every activity of A finishes at or before f_k and a_m starts before f_k, adding a_m would need us to drop something, and Theorem 9.2 applied to the subproblem S_k says the earliest finisher there — which is the next activity we *will* accept — is already safe, so skipping a_m loses nothing. If $s_m \geq f_k$ then a_m is the earliest-finishing member of S_k (everything between k and m was rejected, and the array is sorted by finish time), so by Corollary 9.3 taking it keeps A maximum-size. Either way the invariant holds at $m + 1$. ✓ **Termination.** The loop ends with $m = n + 1$, so A is a maximum-size compatible subset of $\{a_1, \ldots, a_n\} = S$ — which is what we wanted. ✓

The machine confirms that the two forms agree: the block quoted in §9.4.10 shows `same answer as the recursive form: yes`, and the brute-force sweep over all 2048 subsets in the same run puts the true maximum at 4.

9.4.12 The running time

Both forms of the algorithm run in $\Theta(n)$ time **given sorted input**.

The iterative form. Lines 1–3 are constant time. The **for** loop of line 4 runs exactly $n-1$ times, and its body — a comparison, possibly an append and an assignment — is constant time. Total: $\Theta(n)$.

The recursive form. This needs the observation from §9.4.10. Each call's **while** loop advances m, and m is passed to the next call as the new k, from which the next call resumes at $m+1$. So m increases monotonically from 1 to at most $n+1$ across the entire execution, and the total number of loop iterations over all calls is at most n. Each iteration costs $\Theta(1)$, and the number of calls is at most the number of selections plus one, which is at most $n+1$. Total: $\Theta(n)$.

If the input is *not* sorted, we must sort it first, which costs $O(n\lg n)$ by any of the algorithms of Chapter 5. That dominates, so:

> **THEOREM 9.4.** The activity-selection problem on n activities can be solved in $\Theta(n\lg n)$ time and $\Theta(1)$ extra space beyond the output, by sorting the activities into non-decreasing order of finish time and then running **Greedy-Activity-Selector**.

Compare that with the dynamic program of §9.4.5: $\Theta(n^3)$ time and $\Theta(n^2)$ space. At $n=1000$ that is the difference between about ten thousand operations and about 167 million, and between a few bytes and four megabytes. This is what a proof buys you.

Listing 9.2 measures the shape. The table it printed is **Table 9.3**.

Table 9.3 Sorting plus one greedy sweep, measured by **Listing 9.2** 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

The third column is the diagnostic. If the running time really is $c\cdot n\lg n$ for some constant c, then dividing the measured time by $n\lg n$ must give roughly the same number at every size — and it does: 4.03, 3.89, 4.61, 3.91, 4.25, 4.16, with the wobble attributable to cache effects and to the operating system. Had the algorithm been $\Theta(n^2)$, that column would have doubled at every row.

9.4.13 The complete program

Listing 9.2 is the whole of §9.4 in runnable form: both algorithms, the brute-force check, the four candidate rules with their counterexamples, and the timing run.

Listing 9.2 Activity selection in full: the recursive and iterative greedy selectors, the brute-force check, the four candidate rules with counterexamples for the three wrong ones, and a timing run. Compile with `g++ -std=c++17 -O2 -o activity activity.cpp` and run `./activity`.

```
// activity.cpp — The activity-selection problem.
//   * the 11 activities of the running example, solved by the recursive
//     and the iterative greedy selector;
//   * a brute-force check over all 2^n subsets that 4 really is the maximum;
//   * three plausible greedy rules that are NOT correct, each with the smallest
//     counterexample found by exhaustive search over small instances;
//   * a timing run confirming the Theta(n lg n) shape.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
#include <numeric>
#include <chrono>
#include <random>
#include <cmath>
using namespace std;

struct Act { int id, s, f; };

// ---------- the running example, already sorted by finish time ----------
// index 0 is the sentinel a0 with f0 = 0, so that "s[m] >= f[0]" is always true
static const vector<int> S = { 0, 1, 3, 0, 5, 3, 5, 6, 8, 8,  2, 12 };
static const vector<int> F = { 0, 4, 5, 6, 7, 9, 9,10,11,12, 14, 16 };

// ---------- RECURSIVE-ACTIVITY-SELECTOR, exactly as in Section 9.4.9 --
vector<int> recursiveSelector(const vector<int>& s, const vector<int>& f,
                              int k, int n, int depth = 0) {
    cout << string(2 * depth, ' ') << "RECURSIVE-ACTIVITY-SELECTOR(s, f, "
         << k << ", " << n << ")";
    int m = k + 1;
    while (m <= n && s[m] < f[k]) ++m;            // first activity in S_k to finish
    if (m <= n) {
        cout << "  -> m = " << m << ", take a" << m << "\n";
        vector<int> rest = recursiveSelector(s, f, m, n, depth + 1);
        rest.insert(rest.begin(), m);
        return rest;
    }
    cout << "  -> no activity left, return {}\n";
    return {};
}
```

Listing 9.2 (Continued) *Activity selection in full: the recursive and iterative greedy selectors, the brute-force check, the four candidate rules with counterexamples for the three wrong ones, and a timing run. Compile with g++ -std=c++17 -O2 -o activity activity.cpp and run ./activity.*

```
// ---------- GREEDY-ACTIVITY-SELECTOR, the iterative form ----------
vector<int> greedySelector(const vector<int>& s, const vector<int>& f, int n) {
    vector<int> A = { 1 };                        // A = {a1}
    int k = 1;
    for (int m = 2; m <= n; ++m)
        if (s[m] >= f[k]) { A.push_back(m); k = m; }
    return A;
}

// ---------- brute force over every subset: what IS the maximum? ----------
int bruteForceMax(const vector<Act>& a) {
    int n = (int)a.size(), best = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        vector<int> chosen;
        for (int i = 0; i < n; ++i) if (mask >> i & 1) chosen.push_back(i);
        bool ok = true;
        for (size_t i = 0; i + 1 < chosen.size() && ok; ++i)
            for (size_t j = i + 1; j < chosen.size() && ok; ++j) {
                const Act& x = a[chosen[i]]; const Act& y = a[chosen[j]];
                if (!(x.s >= y.f || y.s >= x.f)) ok = false;   // they overlap
            }
        if (ok) best = max(best, (int)chosen.size());
    }
    return best;
}

// ---------- the four candidate greedy rules ----------
// Each repeatedly picks the best remaining activity by its own criterion, then
// discards everything incompatible with it.
typedef int (*Rule)(const vector<Act>&, const vector<char>&);

int ruleEarliestFinish(const vector<Act>& a, const vector<char>& alive) {
    int best = -1;
    for (size_t i = 0; i < a.size(); ++i) if (alive[i])
        if (best < 0 || a[i].f < a[best].f) best = (int)i;
    return best;
}
int ruleEarliestStart(const vector<Act>& a, const vector<char>& alive) {
    int best = -1;
    for (size_t i = 0; i < a.size(); ++i) if (alive[i])
        if (best < 0 || a[i].s < a[best].s) best = (int)i;
    return best;
}
int ruleShortest(const vector<Act>& a, const vector<char>& alive) {
    int best = -1;
    for (size_t i = 0; i < a.size(); ++i) if (alive[i])
```

Listing 9.2 (Continued) *Activity selection in full: the recursive and iterative greedy selectors, the brute-force check, the four candidate rules with counterexamples for the three wrong ones, and a timing run. Compile with g++ -std=c++17 -O2 -o activity activity.cpp and run ./activity.*

```
        if (best < 0 || (a[i].f - a[i].s) < (a[best].f - a[best].s)) best = (int)i;
    return best;
}
int ruleFewestConflicts(const vector<Act>& a, const vector<char>& alive) {
    int best = -1, bestc = 0;
    for (size_t i = 0; i < a.size(); ++i) if (alive[i]) {
        int c = 0;
        for (size_t j = 0; j < a.size(); ++j) if (alive[j] && j != i)
            if (!(a[i].s >= a[j].f || a[j].s >= a[i].f)) ++c;
        if (best < 0 || c < bestc) { best = (int)i; bestc = c; }
    }
    return best;
}

vector<int> runRule(const vector<Act>& a, Rule r) {
    vector<char> alive(a.size(), 1);
    vector<int> picked;
    for (;;) {
        int i = r(a, alive);
        if (i < 0) break;
        picked.push_back(i);
        for (size_t j = 0; j < a.size(); ++j)
            if (alive[j] && !(a[i].s >= a[j].f || a[j].s >= a[i].f)) alive[j] = 0;
        alive[i] = 0;
    }
    return picked;
}

// The three incorrect rules, each on the smallest instance (found by exhaustive
// search) on which it loses to the optimum.
void showCounterexample(const char* name, Rule r, vector<Act> a) {
    vector<int> p = runRule(a, r);
    int opt = bruteForceMax(a);
    cout << "  " << name << " on an instance with n = " << a.size() << ":\n";
    cout << "      activity :";
    for (auto& x : a) cout << setw(9) << ("a" + to_string(x.id));
    cout << "\n      [s, f)   :";
    for (auto& x : a) cout << setw(9) << ("[" + to_string(x.s) + "," + to_string(x.f) +
")");
    cout << "\n      the rule picks " << p.size() << " activities {";
    sort(p.begin(), p.end());
    for (size_t i = 0; i < p.size(); ++i) { if (i) cout << ", "; cout << "a" << a[p[i]].id;
}
    cout << "}\n      brute force says the optimum is " << opt
         << "  ->  the rule is " << (p.size() < (size_t)opt ? "WRONG" : "ok") << "\n\n";
}
```

Listing 9.2 (Continued) *Activity selection in full: the recursive and iterative greedy selectors, the brute-force check, the four candidate rules with counterexamples for the three wrong ones, and a timing run. Compile with g++ -std=c++17 -O2 -o activity activity.cpp and run ./activity.*

```
int main() {
    int n = 11;
    vector<Act> example(n);
    for (int i = 1; i <= n; ++i) example[i - 1] = { i, S[i], F[i] };

    cout << "=== The 11 activities of the running example ===\n";
    cout << "  i :"; for (int i = 1; i <= n; ++i) cout << setw(4) << i;       cout << "\n";
    cout << " si :"; for (int i = 1; i <= n; ++i) cout << setw(4) << S[i];    cout << "\n";
    cout << " fi :"; for (int i = 1; i <= n; ++i) cout << setw(4) << F[i];    cout <<
"\n\n";

    cout << "=== RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11) ===\n";
    vector<int> rec = recursiveSelector(S, F, 0, n);
    cout << "  result = {";
    for (size_t i = 0; i < rec.size(); ++i) { if (i) cout << ", "; cout << "a" << rec[i]; }
    cout << "}, size " << rec.size() << "\n\n";

    cout << "=== GREEDY-ACTIVITY-SELECTOR(s, f) ===\n";
    vector<int> it = greedySelector(S, F, n);
    cout << "  result = {";
    for (size_t i = 0; i < it.size(); ++i) { if (i) cout << ", "; cout << "a" << it[i]; }
    cout << "}, size " << it.size() << "\n";
    cout << "  same answer as the recursive form: "
         << (rec == it ? "yes" : "NO") << "\n\n";

    cout << "=== Brute force over all 2^11 = 2048 subsets ===\n";
    cout << "  true maximum size = " << bruteForceMax(example) << "\n\n";

    cout << "=== Four greedy rules on the running example ===\n";
    struct { const char* name; Rule r; } rules[] = {
        { "earliest finish time ", ruleEarliestFinish },
        { "earliest start time  ", ruleEarliestStart },
        { "shortest duration    ", ruleShortest },
        { "fewest conflicts     ", ruleFewestConflicts },
    };
    for (auto& R : rules) {
        vector<int> p = runRule(example, R.r);
        sort(p.begin(), p.end());
        cout << "  " << R.name << " -> {";
        for (size_t i = 0; i < p.size(); ++i) { if (i) cout << ", "; cout << "a" <<
example[p[i]].id; }
        cout << "}, size " << p.size() << "\n";
    }
    cout << "\n=== Smallest counterexample for each incorrect rule ===\n";
    showCounterexample("earliest start time", ruleEarliestStart,
        { {1,0,8}, {2,1,4}, {3,0,1} });
    showCounterexample("shortest duration ", ruleShortest,
        { {1,5,8}, {2,1,5}, {3,4,6} });
```

Listing 9.2 (Continued) *Activity selection in full: the recursive and iterative greedy selectors, the brute-force check, the four candidate rules with counterexamples for the three wrong ones, and a timing run. Compile with g++ -std=c++17 -O2 -o activity activity.cpp and run ./activity.*

```
    showCounterexample("fewest conflicts  ", ruleFewestConflicts,
        { {1,4,7}, {2,9,13}, {3,2,5}, {4,1,4}, {5,1,2},
          {6,6,10}, {7,0,2}, {8,7,15}, {9,10,11} });

    cout << "=== Timing: sort dominates, so the shape is n lg n ===\n";
    cout << setw(12) << "n" << setw(16) << "time (ms)" << setw(16) << "t / (n lg n)" <<
"\n";
    mt19937 rng(7);
    for (long long N = 250000; N <= 8000000; N *= 2) {
        vector<Act> a((size_t)N);
        for (long long i = 0; i < N; ++i) {
            int x = (int)(rng() % 100000000), d = 1 + (int)(rng() % 1000);
            a[(size_t)i] = { (int)i, x, x + d };
        }
        auto t0 = chrono::steady_clock::now();
        sort(a.begin(), a.end(), [](const Act& p, const Act& q) { return p.f < q.f; });
        int cnt = 0, last = -1;
        for (auto& x : a) if (x.s >= last) { ++cnt; last = x.f; }
        auto t1 = chrono::steady_clock::now();
        double ms = chrono::duration<double, milli>(t1 - t0).count();
        double nlgn = (double)N * (log((double)N) / log(2.0));
        cout << setw(12) << N << setw(16) << fixed << setprecision(2) << ms
             << setw(16) << setprecision(4) << (ms * 1e6 / nlgn)
             << "   (selected " << cnt << ")\n";
    }
    return 0;
}
```

9.5 The Elements of the Greedy Strategy

We have now solved one problem greedily and proved it correct. This section extracts the method, so that you can apply it to a problem you have never seen.

9.5.1 The six-step procedure

The procedure is best stated as a numbered list, because it is the checklist to work through for any candidate greedy algorithm.

> **THE GREEDY STRATEGY IN SIX STEPS. 1.** Determine the **optimal substructure** of the problem. **2.** Develop a **recursive solution**. **3.** Show that if we make the greedy choice, then **only one subproblem remains**. **4.** Prove that it is always **safe** to make the greedy choice. (Steps 3 and 4 can occur in either order.) **5.** Develop a **recursive algorithm** that implements the greedy strategy. **6. Convert** the recursive algorithm to an iterative algorithm.

Check them off against §9.4 and you will find every one:

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**

There is also a shorter, three-line summary of what must be *proved*. It is the version to carry in your head when you solve a new problem against the clock, as Chapter 12 does for problems of exactly this kind:

> **THE THREE THINGS TO PROVE. 1. Cast** the optimization problem as one in which we make a choice and are left with **one** subproblem to solve. **2. Prove** that there is always an optimal solution to the original problem that makes the greedy choice — so the greedy choice is always **safe**. **3. Demonstrate optimal substructure** by showing that, having made the greedy choice, what remains is a subproblem with the property that combining an optimal solution to the subproblem with the greedy choice yields an optimal solution to the original problem.

9.5.2 Ingredient one: the greedy-choice property

> **DEFINITION.** A problem has the **greedy-choice property** if a globally optimal solution can be assembled by making locally optimal (greedy) choices — that is, if for every instance, the choice that the greedy rule makes is contained in **some** optimal solution to that instance.

Three things about this definition repay attention.

"Some", not "every". As §9.4.2 showed, the running example has two different optimal solutions, $\{a_1, a_4, a_8, a_{11}\}$ and $\{a_2, a_4, a_9, a_{11}\}$. The greedy choice a_1 is in the first and not the second. That is fine. All we need is that *at least one* optimal solution agrees with us, because then following that solution's lead cannot hurt.

"Without considering results from subproblems". This point deserves emphasis. The choice may depend on choices made so far — those have already narrowed the problem — but it may not depend on any future choice or on the solution to any subproblem. If your rule needs to know the answer to a subproblem, you are writing a dynamic program, however it looks.

Direction of travel. A greedy algorithm usually progresses in a **top-down** fashion, making one greedy choice after another and reducing each given problem instance to a smaller one. Dynamic programming, by contrast, characteristically works bottom-up (or top-down with memoization, in which case the *code* runs top-down but each subproblem is still solved before the choice above it is made). This is the operational difference: **dynamic programming solves the subproblems before making the first choice; a greedy algorithm makes its first choice before solving any subproblem.**

9.5.3 Ingredient two: optimal substructure

The second ingredient is the one we already know from Chapter 7.

> **DEFINITION (restated from §7.5.2).** A problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems.

Optimal substructure is key in assessing the applicability of dynamic programming **and** of greedy algorithms — it is a shared requirement, not a distinguishing one. But the way we *use* it differs. For a greedy algorithm we assume we arrived at the subproblem by having made the greedy choice in the original problem, and then argue that an optimal solution to that subproblem, combined with the choice already made, yields an optimal solution to the original. That argument implicitly uses **induction on the subproblems** to show that making the greedy choice at every step produces an optimal solution — which is exactly how Corollary 9.3 was used at the end of §9.4.8.

> **BOTH INGREDIENTS ARE NEEDED, AND NEITHER IMPLIES THE OTHER.** The 0-1 knapsack of §9.6 has optimal substructure but **not** the greedy-choice property — which is why it needs a table. The longest-simple-path problem of §7.5.4 has neither. A problem with both is a problem a greedy algorithm solves.

9.5.4 How to recognize the ingredients: a practical recipe

Here is the procedure to follow when you meet a new optimization problem and suspect greed might work.

1. **Write the objective as a single sentence beginning "maximize" or "minimize".** If you cannot, you do not yet have an optimization problem, and neither technique applies.
2. **List every plausible local rule.** In §9.4.3 there were four. Do not stop at your favorite; the trap in §9.4.3 was that two wrong rules got the right answer on one instance.
3. **Try to break each rule on small instances.** Write the brute force. Search exhaustively over instances of size 3, then 4, then 5. A counterexample kills a rule in one line; nothing else does. The search in **Listing 9.2** found a three-activity counterexample to two of the four rules.
4. **For each survivor, attempt the exchange argument.** Take an arbitrary optimal solution; find the element playing the greedy element's role; swap; check feasibility and check the size or cost does not worsen. If the swap breaks feasibility, the rule is probably wrong and step 3 simply has not found the counterexample yet.
5. **Prove optimal substructure by cut-and-paste**, checking the paste is legal (§7.5.3).
6. **Only then write the code.** And keep the brute force: it is your regression test.

> **WHY STEP 3 BEFORE STEP 4.** Searching for a counterexample is cheap, mechanical, and can be done by computer. Searching for a proof is expensive, creative, and can be done only by you. Spend the computer's time first. A great many hours have been lost trying to prove things that are false.

9.6 Greedy versus Dynamic Programming: the Two Knapsacks

This section puts greedy and dynamic programming side by side using a pair of problems so similar that they differ in a single word — and that single word decides which technique works.

> **THE STANDARD MISTAKE.** Both strategies exploit optimal substructure, so it is easy to write a dynamic-programming solution to a problem when a greedy solution suffices or, conversely, to think that a greedy solution works when in fact dynamic programming is required. The two knapsacks are the cleanest illustration in all of algorithms.

9.6.1 The two problems

> **DEFINITION (the 0-1 knapsack problem).** A thief robbing a shop finds n items. Item i is worth v_i rupiah and weighs w_i kilograms, where v_i and w_i are positive integers. The thief's

knapsack holds at most W kilograms. Which items should be taken so as to maximize the total value carried? Each item must be taken **whole or left behind** — hence *0-1*: item i is taken zero times or one time.

DEFINITION (the fractional knapsack problem). Exactly the same, except that the goods are **divisible**: the thief may take any fraction $x_i \in [0,1]$ of item i, obtaining value $x_i v_i$ at weight $x_i w_i$. Maximize $\sum_i x_i\, v_i$ subject to $\sum_i x_i\, w_i \leq W$.

The physical difference is whether the loot is gold bars (0-1: you cannot saw a bar in half without ruining it — or, more honestly, whether the problem *says* you cannot) or gold dust (fractional: scoop out as much as you like).

Both problems have optimal substructure, and the arguments are worth seeing because they are almost identical.

0-1. Consider the most valuable load weighing at most W. If we remove item j from that load, the remainder must be the most valuable load weighing at most $W - w_j$ that can be made from the other $n - 1$ items. *Proof:* if a better load existed for that reduced capacity and reduced item set, substituting it and putting j back would give a legal load of higher value — cut and paste, contradiction. ■

Fractional. Consider the most valuable load weighing at most W. If we remove a weight ω of item j from that load, the remainder must be the most valuable load weighing at most $W - \omega$ that can be made from the other $n - 1$ items plus the remaining $w_j - \omega$ kilograms of item j. Same cut-and-paste argument. ■

So optimal substructure does not distinguish them. What distinguishes them is the greedy-choice property.

9.6.2 The greedy rule, and the instance

The natural greedy rule for either problem is to rank items by **value per kilogram**:

$$\text{density}(i) = \frac{v_i}{w_i}, \tag{9.6}$$

and take the densest first, that is, the item with the largest value of Equation (9.6). The instance this section rests on is the standard three-item one, and **Table 9.5** sets it out.

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

Note that $w_1 + w_2 + w_3 = 10 + 20 + 30 = 60 > 50 = W$, so not everything fits; a choice must be made.

9.6.3 Fractional: greedy succeeds

Run the rule. Capacity left: 50.

- Item 1 is densest at 6.00 per kg. It weighs $10 \leq 50$, so take **all** of it. Value gained 60. Capacity left $50 - 10 = 40$.

- Item 2 is next at 5.00 per kg. It weighs $20 \leq 40$, so take **all** of it. Value gained 100. Capacity left $40 - 20 = 20$.
- Item 3 is next at 4.00 per kg. It weighs $30 > 20$, so take the fraction $x_3 = 20/30 = 2/3$. Value gained $\frac{2}{3} \times 120 = 80$. Capacity left 0.

Total value: $60 + 100 + 80 = 240$. The knapsack is exactly full: $10 + 20 + 20 = 50$ kg.

Now the proof that this is optimal.

> **THEOREM 9.5.** For the fractional knapsack problem, the greedy algorithm — sort the items into non-increasing order of v_i/w_i and take as much of each in turn as the remaining capacity allows — produces an optimal solution.

Proof (exchange argument). Index the items so that $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$.

1. First observe that **any** optimal solution fills the knapsack completely, provided $\sum_i w_i \geq W$. For if $\sum_i x_i\, w_i < W$ and some item is not fully taken, we could increase that item's fraction slightly, gaining positive value — so the solution was not optimal. Hence every optimal solution has $\sum_i x_i\, w_i = W$, and so does the greedy one.
2. Let $x = (x_1, \ldots, x_n)$ be the greedy solution and $y = (y_1, \ldots, y_n)$ be **any** optimal solution. If $x = y$ we are done.
3. Otherwise, let j be the **smallest** index at which they differ. We claim $y_j < x_j$. Indeed, by construction greedy takes the largest fraction of item j that the remaining capacity permits after items $1, \ldots, j-1$; since y agrees with x on items $1, \ldots, j-1$, it has the same remaining capacity at step j, so it cannot take more of item j than greedy did. Hence $y_j \leq x_j$, and since they differ, $y_j < x_j$.
4. Let $\delta = (x_j - y_j)w_j > 0$: the weight of item j that y declined to take and greedy took. Because y fills the knapsack (step 1) and takes less of item j, it must take that weight δ from items with index greater than j. (It cannot take it from items $1, \ldots, j-1$, on which y already agrees with x.)
5. **The exchange.** Construct y' from y by increasing y_j until it equals x_j — that is, adding weight δ of item j — and removing a total weight δ from items of index greater than j, in any way that keeps every $y'_i \in [0,1]$. This is possible by step 4.
6. **The value changes by**

$$\text{value}(y') - \text{value}(y) = \delta \cdot \frac{v_j}{w_j} - \sum_{i>j} \delta_i \cdot \frac{v_i}{w_i}, \tag{9.7}$$

where $\delta_i \geq 0$ is the weight removed from item i and $\sum_{i>j} \delta_i = \delta$. 7. Since $i > j$ implies $v_i/w_i \leq v_j/w_j$,

$$\sum_{i>j} \delta_i \frac{v_i}{w_i} \leq \sum_{i>j} \delta_i \frac{v_j}{w_j} = \delta \cdot \frac{v_j}{w_j}.$$

8. Substituting into Equation (9.7), $\text{value}(y') - \text{value}(y) \geq 0$: the exchange does not lose value. And y' is feasible, because its total weight is unchanged at W and every fraction remains in $[0,1]$.
9. So y' is also optimal, and it agrees with the greedy solution on items $1, \ldots, j$ — one index further than y did.
10. Repeating steps 3–9 at most n times turns any optimal solution into the greedy solution without ever losing value. Therefore the greedy solution is optimal. ■

WHERE THE PROOF USED DIVISIBILITY. In step 5. We needed to add *exactly* weight δ of item j and remove *exactly* weight δ from later items. In a divisible world we can always do that. In the 0-1 world we cannot: items come in lumps, and no lump may be the right size. That single sentence is the whole difference between the two problems.

9.6.4 0-1: greedy fails

Run the same rule, but now items must be taken whole. Capacity left: 50.

- Item 1, density 6.00: weighs $10 \leq 50$. Take it. Value 60. Capacity left 40.
- Item 2, density 5.00: weighs $20 \leq 40$. Take it. Value 160 so far. Capacity left 20.
- Item 3, density 4.00: weighs $30 > 20$. **Cannot take it.** Skip.

Greedy's answer: items 1 and 2, total weight 30 kg, total value **160** — with **20 kg of capacity left empty and unusable**.

Now enumerate every subset, which we can do because there are only $2^3 = 8$ of them. **Table 9.6** is the verified output of **Listing 9.3**.

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

The optimum is {2,3}: weight $20 + 30 = 50$, exactly full, value $100 + 120 = \mathbf{220}$. Greedy returned 160. The shortfall is

$$220 - 160 = 60, \qquad \frac{60}{220} = 0.2727\ldots \approx 27.3\% \text{ of the optimum.}$$

WHY GREED FAILED, PRECISELY. The greedy rule took item 1 because it has the best density. But taking item 1 consumed 10 kg and, worse, left a *remainder* of 40 kg into which items 2 and 3 (weighing 20 and 30) will not both fit. The optimal solution *sacrifices* the densest item in order to make the capacity divide evenly. Greed cannot see that, because seeing it requires looking at what the remaining capacity can hold — which is a subproblem, and a greedy rule is forbidden to look at subproblems.

Note especially that greedy's answer left the knapsack **only 60% full**. That is the tell-tale sign of 0-1 greedy failure: unused capacity that a different, individually-worse choice would have filled.

9.6.5 How badly, and how often?

One counterexample proves the rule is wrong. **Listing 9.3** measures how wrong, over many random instances, with capacity set to half the total weight:

```
=== 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
```

Two lessons hide in those numbers, and both are important.

First, greedy 0-1 is wrong often. At $n = 20$ it was suboptimal in nearly half of twenty thousand random instances. This is not a rare pathology.

Second — and this is the subtle one — greedy 0-1 is usually wrong by very little. The mean ratio of greedy value to optimal value is 0.9789, 0.9868, 0.9930: greedy typically gets within one or two percent. That is precisely why the mistake survives in production code. It gives answers that *look* fine. But the worst case at $n = 5$ was 0.4577 — less than **half** the optimum. If you cannot tolerate that, you cannot use greed.

> **"IT WORKS ON MY TEST DATA" IS NOT A PROOF.** A wrong greedy algorithm whose average error is 1% will pass any casual test suite you write. It will then lose 54% of the value on the one instance that matters. This is the practical reason the proof is the whole job.

9.6.6 The two techniques, side by side

Table 9.7 is the summary to memorize.

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, $\Theta(nW)$

> **THE DIAGNOSTIC QUESTION.** Ask: *can I name the right choice without solving anything?* For the fractional knapsack the answer is yes — take the densest item, because any divisible exchange with a less dense item can only lose value. For the 0-1 knapsack the answer is no:

> whether item 1 belongs in the load depends on how the remaining capacity divides among the other items, which is a subproblem. If the answer is no, build the table.

9.6.7 The complete program

Listing 9.3 The two knapsacks: greedy by density (optimal for the fractional problem, wrong for the 0-1 problem), the 0-1 optimum by dynamic programming, brute force as a check, and a stress test over 120 000 random instances. Compile with `g++ -std=c++17 -O2 -o knapsack knapsack.cpp` and run `./knapsack`.

```
// knapsack.cpp — the two knapsacks side by side.
//   * the fractional knapsack, solved greedily by value per kilogram;
//   * the 0-1 knapsack, where the SAME greedy rule is wrong;
//   * the 0-1 optimum by dynamic programming and by brute force, as a check;
//   * a stress test measuring how often, and by how much, greedy is wrong.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
#include <random>
using namespace std;

struct Item { int id, w, v; };

double density(const Item& it) { return (double)it.v / (double)it.w; }

// ---------- fractional knapsack: greedy by density, correct ----------
struct FracResult { double value; vector<double> frac; };
FracResult fractionalGreedy(vector<Item> a, int W) {
    int n = (int)a.size();
    vector<int> order(n);
    for (int i = 0; i < n; ++i) order[i] = i;
    sort(order.begin(), order.end(),
         [&](int p, int q) { return density(a[p]) > density(a[q]); });
    FracResult r{ 0.0, vector<double>(n, 0.0) };
    int left = W;
    for (int i : order) {
        if (left <= 0) break;
        if (a[i].w <= left) { r.frac[i] = 1.0; r.value += a[i].v; left -= a[i].w; }
        else { r.frac[i] = (double)left / a[i].w; r.value += r.frac[i] * a[i].v; left = 0;
}
    }
    return r;
}

// ---------- 0-1 knapsack: the same greedy rule, applied whole-item ----------
struct IntResult { int value, weight; vector<int> take; };
IntResult zeroOneGreedy(vector<Item> a, int W) {
    int n = (int)a.size();
    vector<int> order(n);
    for (int i = 0; i < n; ++i) order[i] = i;
    sort(order.begin(), order.end(),
         [&](int p, int q) { return density(a[p]) > density(a[q]); });
    IntResult r{ 0, 0, {} };
    int left = W;
    for (int i : order)
```

Listing 9.3 (Continued) *The two knapsacks: greedy by density (optimal for the fractional problem, wrong for the 0-1 problem), the 0-1 optimum by dynamic programming, brute force as a check, and a stress test over 120 000 random instances. Compile with g++ -std=c++17 -O2 -o knapsack knapsack.cpp and run ./knapsack.*

```
        if (a[i].w <= left) { r.take.push_back(i); r.value += a[i].v; r.weight += a[i].w;
left -= a[i].w; }
    return r;
}

// ---------- 0-1 knapsack: the true optimum, by dynamic programming ----------
IntResult zeroOneDP(const vector<Item>& a, int W) {
    int n = (int)a.size();
    vector<vector<int>> best(n + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= n; ++i)
        for (int c = 0; c <= W; ++c) {
            best[i][c] = best[i - 1][c];                          // leave item i out
            if (a[i - 1].w <= c)
                best[i][c] = max(best[i][c], best[i - 1][c - a[i - 1].w] + a[i - 1].v);
        }
    IntResult r{ best[n][W], 0, {} };
    int c = W;
    for (int i = n; i >= 1; --i)
        if (best[i][c] != best[i - 1][c]) {                       // item i was taken
            r.take.push_back(i - 1); r.weight += a[i - 1].w; c -= a[i - 1].w;
        }
    reverse(r.take.begin(), r.take.end());
    return r;
}

// ---------- 0-1 knapsack: brute force, to check the dynamic program ----------
int zeroOneBrute(const vector<Item>& a, int W) {
    int n = (int)a.size(), best = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        int w = 0, v = 0;
        for (int i = 0; i < n; ++i) if (mask >> i & 1) { w += a[i].w; v += a[i].v; }
        if (w <= W) best = max(best, v);
    }
    return best;
}

string names(const vector<int>& idx, const vector<Item>& a) {
    string s;
    for (size_t i = 0; i < idx.size(); ++i) { if (i) s += " + "; s += "item " +
to_string(a[idx[i]].id); }
    return s.empty() ? "(nothing)" : s;
}

int main() {
    // The three-item instance of Section 9.6.2.
    vector<Item> a = { {1, 10, 60}, {2, 20, 100}, {3, 30, 120} };
    int W = 50;
```

Listing 9.3 (Continued) *The two knapsacks: greedy by density (optimal for the fractional problem, wrong for the 0-1 problem), the 0-1 optimum by dynamic programming, brute force as a check, and a stress test over 120 000 random instances. Compile with g++ -std=c++17 -O2 -o knapsack knapsack.cpp and run ./knapsack.*

```
    cout << "=== The instance ===\n";
    cout << setw(8) << "item" << setw(10) << "weight" << setw(10) << "value"
         << setw(16) << "value per kg\n";
    for (auto& it : a)
        cout << setw(8) << it.id << setw(10) << it.w << setw(10) << it.v
             << setw(16) << fixed << setprecision(2) << density(it) << "\n";
    cout << "  knapsack capacity W = " << W << " kg\n\n";

    cout << "=== FRACTIONAL knapsack: greedy by value per kilogram ===\n";
    FracResult fr = fractionalGreedy(a, W);
    for (size_t i = 0; i < a.size(); ++i)
        cout << "  item " << a[i].id << ": take " << setprecision(4) << fr.frac[i]
             << " of it  ->  " << setprecision(2) << fr.frac[i] * a[i].w << " kg, "
             << fr.frac[i] * a[i].v << " in value\n";
    cout << "  TOTAL VALUE = " << setprecision(2) << fr.value << "   (this is
optimal)\n\n";

    cout << "=== 0-1 knapsack: the same greedy rule, whole items only ===\n";
    IntResult g = zeroOneGreedy(a, W);
    cout << "  greedy takes  " << names(g.take, a) << "\n";
    cout << "  weight " << g.weight << " kg of " << W << " (so " << W - g.weight
         << " kg of capacity is wasted), value " << g.value << "\n\n";

    cout << "=== 0-1 knapsack: the true optimum ===\n";
    IntResult d = zeroOneDP(a, W);
    cout << "  optimum takes " << names(d.take, a) << "\n";
    cout << "  weight " << d.weight << " kg of " << W << ", value " << d.value << "\n";
    cout << "  brute force over all 2^3 = 8 subsets agrees: " << zeroOneBrute(a, W) <<
"\n";
    cout << "  greedy loses " << d.value - g.value << " units of value ("
         << setprecision(1) << 100.0 * (d.value - g.value) / d.value << "% of the
optimum)\n\n";

    cout << "=== Every subset of the 0-1 instance, priced ===\n";
    cout << setw(22) << "subset" << setw(10) << "weight" << setw(10) << "value"
         << setw(10) << "fits?\n";
    for (int mask = 0; mask < 8; ++mask) {
        int w = 0, v = 0; string s;
        for (int i = 0; i < 3; ++i) if (mask >> i & 1) {
            w += a[i].w; v += a[i].v; if (!s.empty()) s += "+"; s += to_string(a[i].id);
        }
        if (s.empty()) s = "{}";
        cout << setw(22) << s << setw(10) << w << setw(10) << v
             << setw(10) << (w <= W ? "yes" : "no") << "\n";
    }
```

Listing 9.3 (Continued) *The two knapsacks: greedy by density (optimal for the fractional problem, wrong for the 0-1 problem), the 0-1 optimum by dynamic programming, brute force as a check, and a stress test over 120 000 random instances. Compile with g++ -std=c++17 -O2 -o knapsack knapsack.cpp and run ./knapsack.*

```
    cout << "\n=== Stress test: how often is the greedy 0-1 answer wrong? ===\n";
    mt19937 rng(20260919);
    for (int n : {5, 10, 20}) {
        int trials = 20000, wrong = 0; double worst = 0.0, sumRatio = 0.0;
        for (int t = 0; t < trials; ++t) {
            vector<Item> b(n);
            int total = 0;
            for (int i = 0; i < n; ++i) {
                b[i] = { i + 1, 1 + (int)(rng() % 40), 1 + (int)(rng() % 100) };
                total += b[i].w;
            }
            int cap = total / 2;
            int gv = zeroOneGreedy(b, cap).value;
            int ov = zeroOneDP(b, cap).value;
            if (gv < ov) ++wrong;
            double ratio = ov ? (double)gv / ov : 1.0;
            sumRatio += ratio;
            worst = (t == 0) ? ratio : min(worst, ratio);
        }
        cout << "  n = " << setw(3) << n << ": greedy was suboptimal in " << setw(6) <<
wrong
             << " of " << trials << " random instances ("
             << setprecision(1) << 100.0 * wrong / trials << "%),"
             << "  mean value/optimum = " << setprecision(4) << sumRatio / trials
             << ",  worst = " << worst << "\n";
    }

    cout << "\n=== Fractional greedy is never wrong: same stress test ===\n";
    for (int n : {5, 10, 20}) {
        int trials = 20000, wrong = 0;
        for (int t = 0; t < trials; ++t) {
            vector<Item> b(n);
            int total = 0;
            for (int i = 0; i < n; ++i) {
                b[i] = { i + 1, 1 + (int)(rng() % 40), 1 + (int)(rng() % 100) };
                total += b[i].w;
            }
            int cap = total / 2;
            double fv = fractionalGreedy(b, cap).value;
            // the fractional optimum is at least the 0-1 optimum, and greedy attains it
            if (fv + 1e-9 < zeroOneDP(b, cap).value) ++wrong;
        }
        cout << "  n = " << setw(3) << n << ": fractional greedy fell below the 0-1 optimum
in "
             << wrong << " of " << trials << " instances\n";
    }
    return 0;
}
```

Its first three sections printed exactly the numbers worked by hand above:

```
=== FRACTIONAL knapsack: greedy by value per kilogram ===
  item 1: take 1.0000 of it  ->  10.00 kg, 60.00 in value
  item 2: take 1.0000 of it  ->  20.00 kg, 100.00 in value
  item 3: take 0.6667 of it  ->  20.00 kg, 80.00 in value
  TOTAL VALUE = 240.00   (this is optimal)

=== 0-1 knapsack: the same greedy rule, whole items only ===
  greedy takes  item 1 + item 2
  weight 30 kg of 50 (so 20 kg of capacity is wasted), value 160

=== 0-1 knapsack: the true optimum ===
  optimum takes item 2 + item 3
  weight 50 kg of 50, value 220
  brute force over all 2^3 = 8 subsets agrees: 220
  greedy loses 60 units of value (27.3% of the optimum)
```

Observe finally that the fractional optimum, 240, is strictly greater than the 0-1 optimum, 220. That is no accident: allowing fractions can only help, so the fractional value is always an upper bound on the 0-1 value. That bound is the basis of the branch-and-bound algorithms used to solve large 0-1 knapsack instances in practice [3].

9.7 Huffman Codes

Data compression is the classic application of the greedy method, and Huffman codes compress data very effectively: savings of 20% to 90% are typical [4]. This section builds them completely: the problem, the running example, the trees, the algorithm, the hand trace, the correctness proof, the running time, and a working encoder and decoder.

9.7.1 The problem

We have a file made of characters. We want to store it using fewer bits than the obvious method would need, and we want to be able to get the original file back exactly — **lossless** compression.

The obvious method is a **fixed-length code**: decide how many bits each character will take, and give every character a distinct bit pattern of that length. With $|C|$ distinct characters you need $\lceil \lg|C| \rceil$ bits each, because k bits can name only 2^k different things.

The idea that beats it is a **variable-length code**: give the characters that appear *often* short codewords, and the characters that appear *rarely* long ones. If "a" makes up 45% of the file, spending one bit on each "a" instead of three saves two bits forty-five thousand times over.

> **DEFINITION.** Let C be an alphabet of characters, and for each $c \in C$ let $c.freq$ be the number of times c occurs in the file. A **binary character code** assigns to each $c \in C$ a **codeword**, a finite string of bits. The code is **fixed-length** if all codewords have the same length, and **variable-length** otherwise.

9.7.2 The running example, with the bit counts computed

Figure 9.7 is the table of frequencies and codewords, and it is the instance we shall use for the rest of the section.

	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.

Let us verify both of those numbers ourselves, because the arithmetic is the point.

The alphabet is $C = \{a, b, c, d, e, f\}$, so $|C| = 6$, and the frequencies (in thousands) are 45,13,12,16,9,5. Their sum is

$$45 + 13 + 12 + 16 + 9 + 5 = 100,$$

so the file has 100 thousand $=$100 000 characters. Good — the data is consistent.

The fixed-length code. Six characters need $\lceil \lg 6 \rceil$ bits. Now $\lg 4 = 2$ and $\lg 8 = 3$, so $\lg 6$ lies strictly between 2 and 3, and the ceiling is 3. Three bits give eight patterns, of which we use six: 000,001,010,011,100,101. Every one of the 100 000 characters costs 3 bits, so the file costs

$$3 \times 100\,000 = 300\,000 \text{ bits.}$$

The variable-length code. The table in **Figure 9.7** gives $a = 0$, $b = 101$, $c = 100$, $d = 111$, $e = 1101$, $f = 1100$. Multiply each frequency by its codeword length and add, as **Table 9.8** does.

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**

Adding the last column:

$$45 + 39 + 36 + 48 + 36 + 20 = 224 \text{ thousand bits} = 224\,000 \text{ bits.}$$

The saving is

$$300\,000 - 224\,000 = 76\,000 \text{ bits,} \qquad \frac{76\,000}{300\,000} = 0.2533\ldots = 25.33\%.$$

Another way to say the same thing: the average codeword length is $224\,000/100\,000 = 2.24$ bits per character, against 3 for the fixed code.

> **WHERE THE SAVING COMES FROM.** Look only at the letter "a". It occurs 45 000 times and its codeword shrank from 3 bits to 1, saving $2 \times 45\,000 = 90\,000$ bits all by itself. The letters e and f got *longer* — from 3 bits to 4 — costing $1 \times 9\,000 + 1 \times 5\,000 = 14\,000$ bits. Net saving: $90\,000 - 14\,000 = 76\,000$. ✓ The trade was: make the common thing cheap and pay for it on the rare things.

9.7.3 Prefix-free codes, and why they matter

A variable-length code raises a problem the fixed-length code never had: **where does one codeword stop and the next begin?** With three bits each, you chop the stream into threes. With varying lengths there is nothing to chop by.

Consider the bad code $a = 0$, $b = 01$, $c = 1$. Now decode the bit stream `001`. Is it 0 | 0 | 1, that is "aac"? Or is it 0 | 01, that is "ab"? Both readings are legal. The code is **ambiguous** and therefore useless: we have compressed the file into something we cannot read back.

The trouble is that the codeword for a, namely `0`, is a **prefix** of the codeword for b, namely `01`. Having read a `0`, the decoder cannot tell whether it has finished a character or is halfway through one.

> **DEFINITION.** A code is **prefix-free** (the older literature says *prefix code*) if no codeword is a prefix of any other codeword.

> **WHY PREFIX-FREE IS ENOUGH.** In a prefix-free code, decoding is unambiguous and can be done greedily, left to right, with no look-ahead at all: read bits until what you have read matches a codeword, emit that character, and start again. It must match exactly one codeword, because if the bits you have read matched two codewords, one would be a prefix of the other. That is why the concatenation of codewords needs no separators, no lengths and no escape characters.

Check it on the code of **Figure 9.7** and the stream `0` `101` `100` `111` `1101` `1100`, concatenated as

$$0\,101\,100\,111\,1101\,1100 = 010110011111011100.$$

Decode from the left. `0` — is that a codeword? Yes, "a". Emit "a". Next `1` — a codeword? No. `10` — no. `101` — yes, "b". Emit "b". Next `1` — no. `10` — no. `100` — yes, "c". And so on, recovering `abcdef` with no guesswork.

It is worth knowing — though we shall not prove it — that restricting ourselves to prefix-free codes costs us nothing: for any uniquely-decodable code there is a prefix-free code with exactly the same codeword lengths. So the optimal prefix-free code is the optimal code, full stop.

9.7.4 Codes as binary trees

Prefix-free codes have a beautiful picture. Draw a binary tree. Label every left edge `0` and every right edge `1`. Put each character at a **leaf**. Then the codeword for a character is the sequence of edge labels on the path from the root down to its leaf.

Why is such a code automatically prefix-free? Because a codeword is a prefix of another exactly when one leaf is an ancestor of another — and leaves have no descendants. The tree *enforces* the property.

Figure 9.8 draws the two codes of **Figure 9.7** as trees.

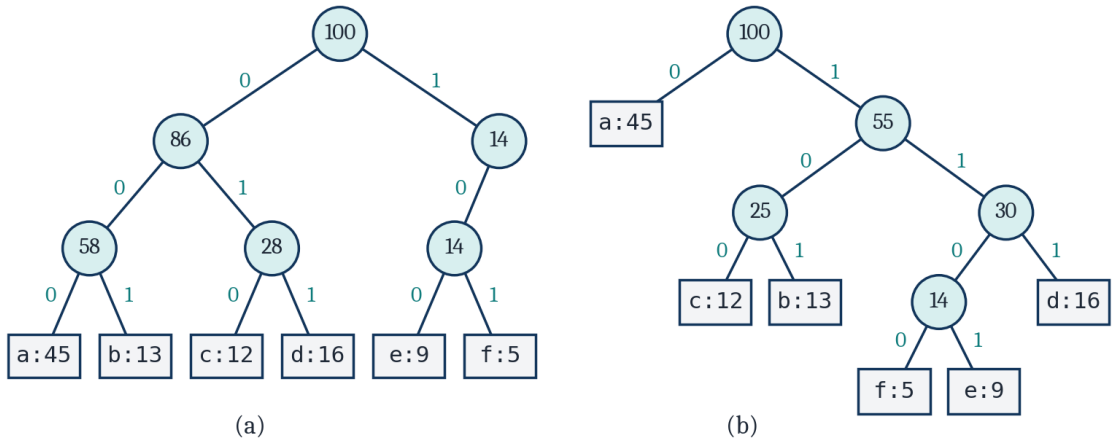


Figure 9.8 Trees for the two coding schemes of **Figure 9.7**. Each leaf carries a character and its frequency; each internal node carries the sum of the frequencies in its subtree. **(a)** the fixed-length code $a = 000, \ldots, f = 101$; **(b)** the optimal prefix code $a = 0, b = 101, \ldots, f = 1100$.

Read tree (b) and recover the codewords. From the root labeled 100, the left edge (`0`) leads straight to the leaf `a:45` — so $a = 0$, one bit. The right edge (`1`) leads to the internal node 55. From 55, left (`0`) goes to 25 and right (`1`) goes to 30. From 25, left goes to `c:12` and right to `b:13` — so $c = 100$ and $b = 101$. From 30, left goes to 14 and right to `d:16` — so $d = 111$. From 14, left goes to `f:5` and right to `e:9` — so $f = 1100$ and $e = 1101$. Every codeword matches **Figure 9.7**.

Two structural observations about tree (b), both important later.

First, the internal node labels. Each internal node is labeled with the sum of the frequencies of the leaves beneath it. Check the root: $45 + 55 = 100$ ✓, and $55 = 25 + 30 = (12 + 13) + (14 + 16) = 25 + 30$ ✓, and $14 = 5 + 9$ ✓. So the root's label is always the total number of characters in the file.

Second, both trees are *full*. A binary tree is **full** if every internal node has exactly two children. Tree (a) is full, and tree (b) is full.

> **LEMMA 9.6.** An optimal prefix-free code for an alphabet C with $|C| \geq 2$ corresponds to a **full** binary tree, which therefore has exactly $|C|$ leaves and exactly $|C| - 1$ internal nodes.

Proof. Suppose some internal node x had exactly one child y. Then delete x and attach y directly to x's parent (or make y the root, if x was the root). Every leaf in y's subtree now has a codeword one bit **shorter**, and no other codeword changes. The tree is still a legal prefix-free code, and its total cost is strictly smaller — so the original tree was not optimal. Hence in an optimal tree every internal node has two children, and the tree is full. A full binary tree with ℓ leaves has exactly $\ell - 1$ internal nodes (proof: each of the $\ell - 1$ merges that builds it creates one internal node; or by induction on ℓ). ■

9.7.5 The cost of a tree

We can now write down, precisely, the thing we want to minimize.

> **DEFINITION.** For a tree T representing a prefix-free code, let $d_T(c)$ denote the **depth** of character c's leaf in T — the number of edges on the path from the root to that leaf, which is exactly the number of bits in c's codeword. The **cost** of T is

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c),$$

(9.8)

the total number of bits needed to encode the whole file. A tree achieving the minimum possible $B(T)$ is an **optimal code** for C.

Every symbol: C is the alphabet; c runs over its characters; $c.freq$ is how often c occurs; $d_T(c)$ is how many bits c's codeword has in this particular tree; the product is how many bits all the copies of c cost; the sum over c is the whole file.

Apply Equation (9.8) to tree (b) of **Figure 9.8**: the depths are $d(a) = 1$, $d(b) = 3$, $d(c) = 3$, $d(d) = 3$, $d(e) = 4$, $d(f) = 4$, so

$$\begin{aligned} B(T) &= 45(1) + 13(3) + 12(3) + 16(3) + 9(4) + 5(4) \\ &= 45 + 39 + 36 + 48 + 36 + 20 = 224, \end{aligned}$$

in thousands of bits — the same 224 000 as **Table 9.8**. And for tree (a), every depth is 3, so $B(T) = 3(45 + 13 + 12 + 16 + 9 + 5) = 3(100) = 300$.

AN EQUIVALENT FORMULA THAT MAKES THE ALGORITHM OBVIOUS. There is a second way to compute $B(T)$ which is the key to everything that follows. Instead of summing over leaves, **sum the labels of the internal nodes**. For tree (b) those labels are 100,55,25,30,14, and

$$100 + 55 + 25 + 30 + 14 = 224.$$

Why does this work? A character c at depth $d_T(c)$ lies beneath exactly $d_T(c)$ internal nodes — one for each edge on its path to the root — and it contributes $c.freq$ to each of their labels. So summing all internal labels counts $c.freq$ exactly $d_T(c)$ times, which is $c.freq \cdot d_T(c)$. Summing over c gives $B(T)$. In other words: **the cost of the tree is the total of all the merges that built it** — which is exactly the quantity the greedy algorithm minimizes, one merge at a time.

9.7.6 The algorithm

Figure 9.9 gives the pseudocode. It is nine lines.

```
    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.

Line by line:

- **Line 1**, `n = |C|`. How many characters there are.
- **Line 2**, `Q = C`. Build a min-priority queue containing one leaf node per character, keyed on frequency. (A *min-priority queue* is the data structure of Chapter 5 that supports "give me the smallest" and "add this" efficiently; a binary min-heap does both in $O(\lg n)$.)
- **Line 3**, `for i = 1 to n-1`. Exactly $n - 1$ iterations — by Lemma 9.6, exactly the number of internal nodes we must create.
- **Line 4**, `allocate a new node z`. This will be an internal node.
- **Line 5**, `z.left = x = Extract-Min(Q)`. Remove the least frequent item and make it z's left child.
- **Line 6**, `z.right = y = Extract-Min(Q)`. Remove the (now) least frequent item and make it z's right child.
- **Line 7**, `z.freq = x.freq + y.freq`. The new node's frequency is the sum.
- **Line 8**, `Insert(Q, z)`. Put the merged tree back into the queue, where it competes on equal terms with the leaves.
- **Line 9**, `return Extract-Min(Q)`. After $n - 1$ merges the queue holds exactly one item — the root of the finished tree.

> **THE GREEDY CHOICE, STATED.** *Merge the two least frequent items.* That is the whole rule. Note how it connects to the insight of §9.7.5: the cost of the tree is the sum of the internal labels, the merge we are about to perform creates an internal label equal to $x.freq + y.freq$, and choosing the two smallest makes that new label as small as it can be. Greed here means: make the cheapest merge available, now.

Note also *why* merging pushes things down. Whenever x and y become children of a new node, every leaf in their subtrees gets one edge further from the root — so its codeword grows by one bit. Merging the two *least frequent* items therefore lengthens the codewords of the characters that can least afford to be lengthened least often. That is the intuition; §9.7.8 turns it into a proof.

9.7.7 The trace, step by step, on the running example

Figure 9.10 is the complete trace.

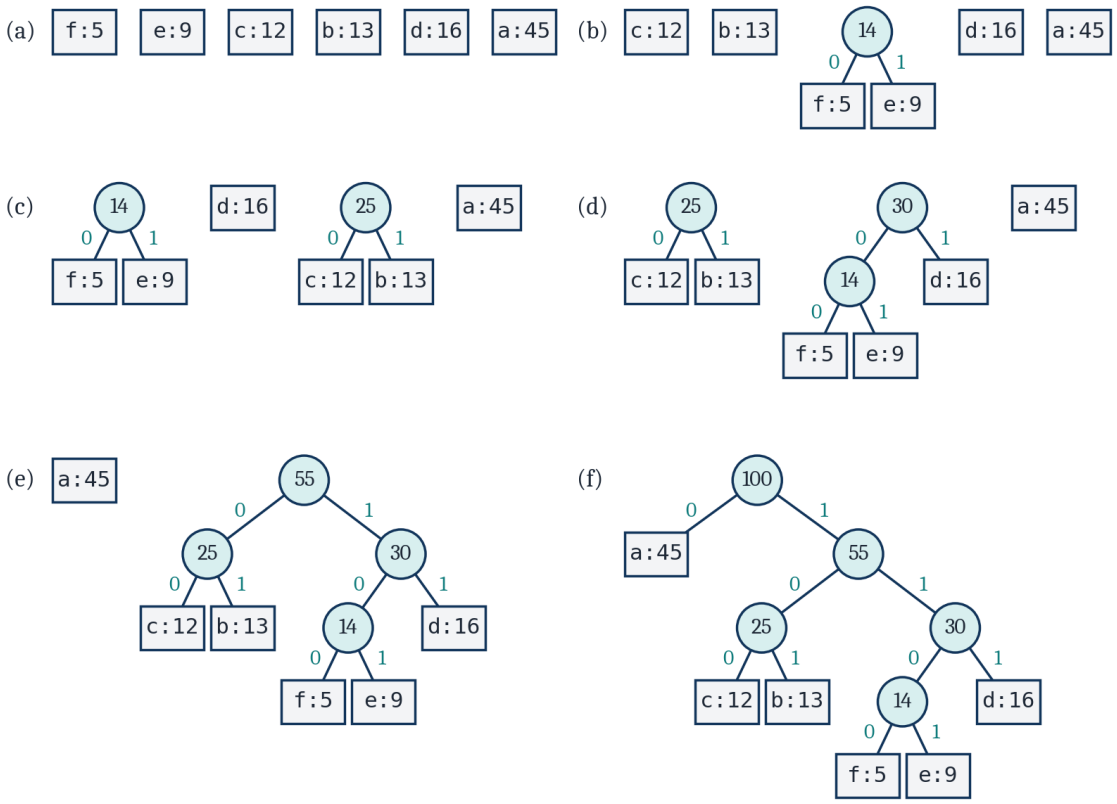


Figure 9.10 The six-panel trace of **Huffman** on the frequencies of **Figure 9.7**. Each panel shows the queue sorted into increasing order of frequency; at each step the two lowest-frequency trees are merged. Leaves are rectangles, internal nodes circles. **(a)** the initial six leaves; **(b)–(e)** the intermediate stages; **(f)** the final tree.

Now the same trace in words and arithmetic. The queue starts, sorted, as

f:5 e:9 c:12 b:13 d:16 a:45

which is panel (a).

Merge 1. The two smallest are f: 5 and e: 9. Extract both. New internal node with frequency $5 + 9 = 14$, left child f, right child e. Insert. The queue is now

c:12 b:13 (14) d:16 a:45

— panel (b). Note that the new node, at 14, slots in *between* b: 13 and d: 16: merged trees and untouched leaves compete purely on frequency.

Merge 2. The two smallest are c: 12 and b: 13. New node $12 + 13 = 25$, left child c, right child b. Queue:

(14) d:16 (25) a:45

— panel (c).

Merge 3. The two smallest are the node (14) and d: 16. New node $14 + 16 = 30$, left child the (14) subtree, right child d. Queue:

(25) (30) a:45

— panel (d). Here f and e have been pushed down a second time; their codewords are now two bits long and will end at four.

Merge 4. The two smallest are (25) and (30). New node $25 + 30 = 55$. Queue:

$$\text{a:45} \quad (55)$$

— panel (e).

Merge 5. The last two: a: 45 and (55). New node $45 + 55 = 100$, left child a, right child the (55) subtree. Queue: (100) — panel (f), the finished tree, which is exactly tree (b) of **Figure 9.8**.

Five merges for six characters: $n - 1 = 5$ ✓, as Lemma 9.6 requires.

And now check the insight of §9.7.5 against the merges we just performed. The internal labels created were 14,25,30,55,100, and

$$14 + 25 + 30 + 55 + 100 = 224.$$

The cost of the tree is the sum of the merges. **Table 9.9** tabulates the whole run.

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**

The program in **Listing 9.4** prints this trace and it agrees line for line:

```
=== Part 2: HUFFMAN(C), traced ===
  (a) start: f:5  e:9  c:12  b:13  d:16  a:45
  merge f:5 + e:9 = 14
        queue now: c:12  b:13  (14)  d:16  a:45
  merge c:12 + b:13 = 25
        queue now: (14)  d:16  (25)  a:45
  merge (14) + d:16 = 30
        queue now: (25)  (30)  a:45
  merge (25) + (30) = 55
        queue now: a:45  (55)
  merge a:45 + (55) = 100
        queue now: (100)

=== Part 3: the codebook read off the tree ===
    char      freq    codeword     depth freq x depth
       a        45           0         1            45
       b        13         101         3            39
       c        12         100         3            36
       d        16         111         3            48
       e         9        1101         4            36
       f         5        1100         4            20
  B(T) = 224 thousand bits = 224000 bits
  cost() walking the tree agrees: 224
```

Every codeword matches the table in **Figure 9.7**.

9.7.8 Correctness, part one: the greedy-choice lemma

Now we prove the algorithm optimal. As §9.5 taught, that means two lemmas: the greedy choice is safe, and what remains has optimal substructure.

> **LEMMA 9.7 (the greedy-choice property for Huffman codes).** Let C be an alphabet in which each character c has frequency $c.freq$, and let x and y be two characters of C having the **lowest** frequencies. Then there exists an optimal prefix-free code for C in which the codewords for x and y have the **same length** and **differ only in the last bit** — equivalently, in which x and y are sibling leaves of maximum depth.

Proof (exchange argument, twice). The idea: take any optimal tree, and swap x and y into the two deepest sibling positions, showing each swap cannot increase the cost.

1. Let T be **any** optimal tree for C. By Lemma 9.6, T is full.
2. Let a and b be two leaves that are **siblings of maximum depth** in T. Such a pair exists: take any deepest leaf; its parent is internal, so by fullness it has another child; that child is a leaf too (otherwise it would have descendants deeper than the deepest leaf). Write d for their common depth, which is the height of T.
3. Without loss of generality assume $x.freq \leq y.freq$ and $a.freq \leq b.freq$. (If not, rename.)
4. Because x and y have the two lowest frequencies in C, and a, b are characters of C, we have

$$x.freq \leq a.freq \qquad \text{and} \qquad y.freq \leq b.freq.$$

(For the first: x has the lowest frequency of all, so it is $\leq$ everything. For the second: y has the second lowest, so $y.freq \leq$ the frequency of any character other than x; and $b \neq x$ because if b were x then, since $a.freq \leq b.freq = x.freq \leq a.freq$, we would have $a.freq = x.freq$ and could rename a as x.) 5. **First exchange.** Build T' from T by **swapping the positions of x and a** — that is, exchange the two leaves, keeping all frequencies attached to their characters. Compute the change in cost. Only the two swapped characters change depth, so

$$\begin{aligned} &B(T) - B(T') \\ &= [x.freq \cdot d_T(x) + a.freq \cdot d_T(a)] - [x.freq \cdot d_{T'}(x) + a.freq \\ &\qquad \cdot d_{T'}(a)]. \end{aligned}$$

6. But $d_{T'}(x) = d_T(a)$ and $d_{T'}(a) = d_T(x)$, since the two simply changed places. Substituting and collecting terms,

$$\begin{aligned} &B(T) - B(T') \\ &= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x), \end{aligned}$$

and grouping the $a.freq$ terms and the $x.freq$ terms, this factors as

$$B(T) - B(T') = (a.freq - x.freq)(d_T(a) - d_T(x)). \tag{9.9}$$

7. Now examine the two factors of Equation (9.9). The first, $a.freq - x.freq$, is ≥ 0 by step 4. The second, $d_T(a) - d_T(x)$, is ≥ 0 because a is a leaf of **maximum** depth, so no leaf — x included — is deeper. A product of two non-negative numbers is non-negative, so

$$B(T) - B(T') \geq 0, \qquad \text{that is,} \qquad B(T') \leq B(T).$$

8. Since T was optimal, $B(T') \geq B(T)$ too; hence $B(T') = B(T)$ and T' is also optimal. The exchange cost us nothing.
9. **Second exchange.** Build T'' from T' by swapping y and b. The identical computation, using $y.freq \leq b.freq$ and the fact that b still has maximum depth in T' (swapping x and a did not change the *set* of depths, only which character sits where), gives $B(T'') \leq B(T')$, and therefore $B(T'') = B(T') = B(T)$: T'' is optimal.
10. In T'', the characters x and y occupy the positions that a and b occupied in T — sibling leaves of maximum depth. So their codewords have the same length d and differ only in the final bit. ■

READ STEP 7 AGAIN. It is the entire proof in one line. *A low-frequency character and a deep position belong together.* If the cheap character is not already in the deep position, swapping them multiplies a non-negative frequency difference by a non-negative depth difference, and the cost can only go down. That is why merging the two rarest symbols first — and thereby pushing them to the bottom of the tree — is safe.

9.7.9 Correctness, part two: the optimal-substructure lemma

LEMMA 9.8 (optimal substructure for Huffman codes). Let C be an alphabet, let $x, y \in C$ be characters of minimum frequency, and form a new alphabet

$$C' = (C \setminus \{x, y\}) \cup \{z\},$$

where z is a new character with $z.freq = x.freq + y.freq$. Let T' be **any** optimal tree for C'. Form T from T' by replacing the leaf for z with an internal node having x and y as children. Then T is an optimal tree for C.

Proof. First we relate the two costs; then we argue by contradiction.

1. **The cost relation.** Every character $c \in C \setminus \{x, y\}$ is a leaf of both trees at the same depth, so $d_T(c) = d_{T'}(c)$ and c contributes identically to both costs.
2. The characters x and y sit one level below where z sat, so $d_T(x) = d_T(y) = d_{T'}(z) + 1$. Their joint contribution to $B(T)$ is

$$x.freq \cdot d_T(x) + y.freq \cdot d_T(y) = (x.freq + y.freq)(d_{T'}(z) + 1)$$

$$= z.freq \cdot d_{T'}(z) + (x.freq + y.freq).$$

3. The first term, $z.freq \cdot d_{T'}(z)$, is exactly what z contributed to $B(T')$. So, subtracting,

$$B(T) = B(T') + x.freq + y.freq. \tag{9.10}$$

In words: **collapsing the pair $\{x, y\}$ into a single symbol z reduces the cost by exactly $x.freq + y.freq$, whatever the tree.** This is the formal version of "the cost is the sum of the merges" from §9.7.5. 4. **The contradiction.** Suppose T is **not** optimal for C. Then some tree T'' has $B(T'') < B(T)$. 5. By Lemma 9.7 we may assume, without loss of generality, that T'' has x and y as sibling leaves — because Lemma 9.7 guarantees an optimal tree with that property exists, and T'' can be taken to be one of minimum cost, hence optimal, hence (after the exchanges of Lemma 9.7, which do not change the cost) of that form. 6. Now go the other way: form T''' from T'' by deleting the leaves x and y and turning their parent into a leaf labeled z with $z.freq = x.freq + y.freq$. Then T''' is a legal tree for the alphabet C', and by exactly the computation of steps 1–3,

$$B(T''') = B(T'') - x.freq - y.freq.$$

7. Substituting the assumption $B(T'') < B(T)$ and then Equation (9.10),

$$B(T''') = B(T'') - x.freq - y.freq < B(T) - x.freq - y.freq = B(T').$$

8. So T''' is a tree for C' with strictly smaller cost than T' — contradicting the assumption that T' was optimal for C'.
9. Hence no such T'' exists, and T is optimal for C. ■

9.7.10 The theorem

THEOREM 9.9. The procedure **Huffman** of **Figure 9.9** produces an optimal prefix-free code.

Proof. By induction on $n = |C|$.

Base case, $n = 2$. The procedure performs one merge, producing a tree with two leaves at depth 1, so both codewords are one bit. No prefix-free code for two characters can do better than one bit each, since a code with a zero-length codeword is not prefix-free when there are two characters. So the output is optimal.

Inductive step. Assume the procedure is optimal on every alphabet of size $n - 1$, and let $|C| = n \geq 3$. Lines 5–7 of the first iteration extract two characters x, y of minimum frequency and replace them by a node z with $z.freq = x.freq + y.freq$. The queue now contains exactly the alphabet C' of Lemma 9.8, of size $n - 1$, and the remaining $n - 2$ iterations are precisely a run of **Huffman** on C'. By the inductive hypothesis, that run returns an optimal tree T' for C'. The full procedure's output is T' with the leaf z replaced by the internal node whose children are x and y — which is exactly the tree T of Lemma 9.8. Lemma 9.8 says T is optimal for C. Lemma 9.7 is what makes Lemma 9.8's hypothesis legitimate: it guarantees that insisting x and y be siblings does not exclude every optimal tree. ■

9.7.11 The running time

Count the operations in **Figure 9.9**.

- **Line 2** builds a priority queue of n elements. Using **Build-Min-Heap** from Chapter 5 this is $O(n)$.
- **Line 3** runs the loop $n - 1$ times.
- **Lines 5 and 6** each perform one `Extract-Min`, at $O(\lg n)$ on a binary heap.
- **Line 8** performs one `Insert`, also $O(\lg n)$.
- **Lines 4 and 7** are $O(1)$.

So the loop costs $(n - 1) \times O(\lg n) = O(n \lg n)$, and the build costs $O(n)$.

THEOREM 9.10. Huffman runs in $O(n \lg n)$ time on an alphabet of n characters, using a binary min-heap for Q.

Two refinements worth knowing. First, if the characters are **already sorted by frequency**, the whole thing can be done in $O(n)$ time with two queues instead of a heap — one holding the sorted leaves, one holding the internal nodes in the order they are created (which is automatically non-decreasing) — and the minimum is always at the front of one of the two. That is Exercise 9.12. Second, a cleverer priority queue does not help in general: when the frequencies can be compared but not otherwise exploited, building an optimal code involves the sort-like work of ordering them, and $\Theta(n \lg n)$ is the best one can do. Integer priority queues such as van Emde Boas trees help only when the frequencies are small integers.

9.7.12 The complete encoder and decoder

Listing 9.4 is a working compressor. It builds the tree, prints the trace and the codebook, computes $B(T)$ both ways as a cross-check, encodes real text into a genuine bit stream packed into bytes, decodes it back, and verifies the result is identical to the input. It also handles the awkward one-symbol case, where the tree would otherwise have a zero-length codeword.

Listing 9.4 A complete Huffman encoder and decoder: the traced construction on the frequencies of **Figure 9.7**, the codebook, $B(T)$ computed two independent ways, a real bit-packed round trip on ordinary text, the one-symbol edge case, and a timing run. Compile with `g++ -std=c++17 -O2 -o huffman huffman.cpp` and run `./huffman`.

```
// huffman.cpp — a complete Huffman encoder and decoder.
//   * builds the tree for the frequency table of Section 9.7.2, printing the
//     queue at every merge so the trace can be checked against the figure;
//   * prints the codeword table and the cost B(T);
//   * compares against the fixed-length code;
//   * then encodes and decodes a real block of text, bit by bit, and checks
//     that what comes out is exactly what went in.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <queue>
#include <map>
#include <algorithm>
#include <cstdint>
#include <cmath>
#include <chrono>
#include <random>
using namespace std;

// ------------------------------------------------------------------ the tree ---
struct Node {
    long long freq;
    char ch;            // meaningful only at a leaf
    bool leaf;
    int serial;         // creation order, used only to break ties reproducibly
    Node *left, *right;
    Node(long long f, char c) : freq(f), ch(c), leaf(true),  serial(0), left(nullptr),
right(nullptr) {}
    Node(Node* l, Node* r)    : freq(l->freq + r->freq), ch(0), leaf(false), serial(0),
left(l), right(r) {}
};

// The priority queue is a MIN-queue on frequency. Ties go to the node that was
// created first, which makes the whole construction deterministic.
struct Later {
    bool operator()(const Node* a, const Node* b) const {
        if (a->freq != b->freq) return a->freq > b->freq;
        return a->serial > b->serial;
    }
};

string label(const Node* n) {
    if (n->leaf) return string(1, n->ch) + ":" + to_string(n->freq);
    return "(" + to_string(n->freq) + ")";
}
```

Listing 9.4 (Continued) *A complete Huffman encoder and decoder: the traced construction on the frequencies of Figure 9.7, the codebook, B(T) computed two independent ways, a real bit-packed round trip on ordinary text, the one-symbol edge case, and a timing run. Compile with g++ -std=c++17 -O2 -o huffman huffman.cpp and run ./huffman.*

```
// Print the whole queue in increasing order, without disturbing it.
void showQueue(priority_queue<Node*, vector<Node*>, Later> q, const string& tag) {
    cout << "  " << tag << " ";
    bool first = true;
    while (!q.empty()) { if (!first) cout << "  "; cout << label(q.top()); q.pop(); first =
false; }
    cout << "\n";
}

// HUFFMAN(C) — the textbook algorithm, with the queue printed at every step.
Node* huffman(const vector<pair<char, long long>>& C, bool trace) {
    priority_queue<Node*, vector<Node*>, Later> Q;
    int serial = 0;
    vector<pair<char, long long>> sorted = C;
    sort(sorted.begin(), sorted.end(),
         [](const pair<char, long long>& a, const pair<char, long long>& b) { return
a.second < b.second; });
    for (auto& p : sorted) { Node* n = new Node(p.second, p.first); n->serial = serial++;
Q.push(n); }
    if (Q.empty()) return nullptr;
    if (Q.size() == 1) {                 // one distinct symbol: give it one bit
        Node* only = Q.top(); Q.pop();
        Node* root = new Node(only, new Node(0, '\0'));
        root->serial = serial++;
        return root;
    }
    int n = (int)Q.size();
    if (trace) showQueue(Q, "(a) start:");
    for (int i = 1; i <= n - 1; ++i) {
        Node* x = Q.top(); Q.pop();      // x = EXTRACT-MIN(Q)
        Node* y = Q.top(); Q.pop();      // y = EXTRACT-MIN(Q)
        Node* z = new Node(x, y);        // z.left = x, z.right = y, z.freq = x.freq +
y.freq
        z->serial = serial++;
        Q.push(z);                       // INSERT(Q, z)
        if (trace) {
            cout << "  merge " << label(x) << " + " << label(y)
                 << " = " << z->freq << "\n";
            showQueue(Q, "      queue now:");
        }
    }
    return Q.top();
}
```

Listing 9.4 (Continued) *A complete Huffman encoder and decoder: the traced construction on the frequencies of Figure 9.7, the codebook, B(T) computed two independent ways, a real bit-packed round trip on ordinary text, the one-symbol edge case, and a timing run. Compile with g++ -std=c++17 -O2 -o huffman huffman.cpp and run ./huffman.*

```
// ------------------------------------------------------------ the codebook --
void collect(const Node* n, string code, map<char, string>& out) {
    if (!n) return;
    if (n->leaf) { out[n->ch] = code.empty() ? "0" : code; return; }
    collect(n->left,  code + "0", out);
    collect(n->right, code + "1", out);
}

// B(T) = sum over characters c of c.freq * d_T(c)
long long cost(const Node* n, int depth = 0) {
    if (!n) return 0;
    if (n->leaf) return n->freq * depth;
    return cost(n->left, depth + 1) + cost(n->right, depth + 1);
}

// ------------------------------------------------------ encoding / decoding ---
// Encode into a real bit stream: bytes plus a count of valid bits.
struct BitStream { vector<uint8_t> bytes; size_t nbits = 0; };

void pushBit(BitStream& bs, int bit) {
    if (bs.nbits % 8 == 0) bs.bytes.push_back(0);
    if (bit) bs.bytes.back() |= (uint8_t)(1u << (7 - (bs.nbits % 8)));
    ++bs.nbits;
}

BitStream encode(const string& text, const map<char, string>& code) {
    BitStream bs;
    for (char c : text) {
        auto it = code.find(c);
        if (it == code.end()) { cerr << "symbol not in codebook\n"; exit(1); }
        for (char b : it->second) pushBit(bs, b == '1');
    }
    return bs;
}

// Decoding walks the tree: 0 go left, 1 go right; on reaching a leaf, emit its
// character and jump back to the root. No length prefix is needed -- that is
// exactly what "prefix-free" buys us.
string decode(const BitStream& bs, const Node* root, size_t nsymbols) {
    string out;
    const Node* cur = root;
    if (root && root->leaf) { return string(nsymbols, root->ch); }
    for (size_t i = 0; i < bs.nbits && out.size() < nsymbols; ++i) {
        int bit = (bs.bytes[i / 8] >> (7 - (i % 8))) & 1;
        cur = bit ? cur->right : cur->left;
        if (cur->leaf) { out += cur->ch; cur = root; }
```

Listing 9.4 (Continued) *A complete Huffman encoder and decoder: the traced construction on the frequencies of Figure 9.7, the codebook, B(T) computed two independent ways, a real bit-packed round trip on ordinary text, the one-symbol edge case, and a timing run. Compile with g++ -std=c++17 -O2 -o huffman huffman.cpp and run ./huffman.*

```
    }
    return out;
}

int main() {
    cout << "=== Part 1: the frequency table of the running example ===\n";
    // frequencies in thousands; the file holds 100 000 characters
    vector<pair<char, long long>> C = {
        {'a', 45}, {'b', 13}, {'c', 12}, {'d', 16}, {'e', 9}, {'f', 5}
    };
    long long total = 0; for (auto& p : C) total += p.second;
    cout << "  characters: ";
    for (auto& p : C) cout << p.first << ":" << p.second << "  ";
    cout << "\n  total = " << total << " thousand characters\n\n";

    cout << "=== Part 2: HUFFMAN(C), traced ===\n";
    Node* root = huffman(C, true);
    cout << "\n";

    cout << "=== Part 3: the codebook read off the tree ===\n";
    map<char, string> code;
    collect(root, "", code);
    cout << setw(8) << "char" << setw(10) << "freq" << setw(12) << "codeword"
         << setw(10) << "depth" << setw(14) << "freq x depth\n";
    long long B = 0;
    for (auto& p : C) {
        int d = (int)code[p.first].size();
        cout << setw(8) << p.first << setw(10) << p.second << setw(12) << code[p.first]
             << setw(10) << d << setw(14) << p.second * d << "\n";
        B += p.second * d;
    }
    cout << "  B(T) = " << B << " thousand bits = " << B * 1000 << " bits\n";
    cout << "  cost() walking the tree agrees: " << cost(root) << "\n\n";

    cout << "=== Part 4: fixed-length against Huffman ===\n";
    int fixedBits = (int)ceil(log2((double)C.size()));
    cout << "  " << C.size() << " distinct characters need ceil(lg " << C.size()
         << ") = " << fixedBits << " bits each\n";
    cout << "  fixed-length total  = " << fixedBits << " x " << total * 1000
         << " = " << fixedBits * total * 1000 << " bits\n";
    cout << "  Huffman total       = " << B * 1000 << " bits\n";
    cout << "  saving              = " << (fixedBits * total - B) * 1000 << " bits = "
         << fixed << setprecision(2)
         << 100.0 * (fixedBits * total - B) / (fixedBits * total) << "%\n";
    cout << "  average codeword length = " << (double)B / total << " bits per
character\n\n";
```

Listing 9.4 (Continued) *A complete Huffman encoder and decoder: the traced construction on the frequencies of Figure 9.7, the codebook, B(T) computed two independent ways, a real bit-packed round trip on ordinary text, the one-symbol edge case, and a timing run. Compile with g++ -std=c++17 -O2 -o huffman huffman.cpp and run ./huffman.*

```
    cout << "=== Part 5: encode and decode real text ===\n";
    string text =
        "abracadabra abracadabra a laundry in surabaya washes bedsheets "
        "and towels and shirts and the dryer never stops running all day";
    map<char, long long> tally;
    for (char c : text) ++tally[c];
    vector<pair<char, long long>> C2(tally.begin(), tally.end());
    Node* root2 = huffman(C2, false);
    map<char, string> code2;
    collect(root2, "", code2);

    cout << "  input length      : " << text.size() << " characters\n";
    cout << "  distinct symbols  : " << C2.size() << "\n";
    cout << "  codebook (symbol -> codeword), shortest first:\n";
    vector<pair<char, string>> cb(code2.begin(), code2.end());
    sort(cb.begin(), cb.end(), [](const pair<char, string>& a, const pair<char, string>& b)
{
        if (a.second.size() != b.second.size()) return a.second.size() < b.second.size();
        return a.first < b.first;
    });
    int perLine = 0;
    cout << "    ";
    for (auto& p : cb) {
        cout << " '" << (p.first == ' ' ? '_' : p.first) << "'=" << p.second;
        if (++perLine % 6 == 0) cout << "\n    ";
    }
    cout << "\n";

    BitStream bs = encode(text, code2);
    string back = decode(bs, root2, text.size());
    cout << "  fixed-length cost : " << text.size() * 8 << " bits (8 bits per char)\n";
    cout << "  Huffman cost      : " << bs.nbits << " bits in "
         << bs.bytes.size() << " bytes\n";
    cout << "  compression       : " << setprecision(2)
         << 100.0 * (1.0 - (double)bs.nbits / (text.size() * 8)) << "% smaller\n";
    cout << "  round trip        : " << (back == text ? "IDENTICAL" : "DIFFERENT") << "\n";
    cout << "  first 64 bits     : ";
    for (size_t i = 0; i < 64 && i < bs.nbits; ++i)
        cout << (((bs.bytes[i / 8] >> (7 - (i % 8))) & 1) ? '1' : '0');
    cout << "\n  decoded prefix    : \"" << back.substr(0, 40) << "...\"\n\n";
```

Listing 9.4 (Continued) *A complete Huffman encoder and decoder: the traced construction on the frequencies of Figure 9.7, the codebook, B(T) computed two independent ways, a real bit-packed round trip on ordinary text, the one-symbol edge case, and a timing run. Compile with g++ -std=c++17 -O2 -o huffman huffman.cpp and run ./huffman.*

```
    cout << "=== Part 6: the one-symbol edge case ===\n";
    string flat(1000, 'z');
    map<char, long long> t3; for (char c : flat) ++t3[c];
    vector<pair<char, long long>> C3(t3.begin(), t3.end());
    Node* root3 = huffman(C3, false);
    map<char, string> code3; collect(root3, "", code3);
    BitStream bs3 = encode(flat, code3);
    cout << "  1000 copies of 'z': codeword \"" << code3['z'] << "\", "
         << bs3.nbits << " bits, round trip "
         << (decode(bs3, root3, flat.size()) == flat ? "IDENTICAL" : "DIFFERENT") <<
"\n\n";

    cout << "=== Part 7: the running time is O(n lg n) ===\n";
    cout << setw(12) << "n symbols" << setw(16) << "build (ms)"
         << setw(18) << "t / (n lg n)\n";
    mt19937 rng(20260919);
    for (int n = 4000; n <= 512000; n *= 2) {
        vector<pair<char, long long>> big(n);
        for (int i = 0; i < n; ++i) big[i] = { (char)(i % 128), 1 + (long long)(rng() %
1000000) };
        auto t0 = chrono::steady_clock::now();
        Node* r = huffman(big, false);
        auto t1 = chrono::steady_clock::now();
        double ms = chrono::duration<double, milli>(t1 - t0).count();
        double nlgn = (double)n * log2((double)n);
        cout << setw(12) << n << setw(16) << setprecision(3) << ms
             << setw(18) << setprecision(5) << ms * 1e6 / nlgn
             << "   (root freq " << r->freq << ")\n";
    }
    return 0;
}
```

Parts 4, 5 and 6 printed:

```
=== 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     : 1110100001111010111111100111010000111110111101000011110101111111
  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
```

Three things to notice in that output.

The saving is real and large. A 126-character sentence took 1008 bits stored plainly and 489 bits Huffman-coded — **51.5% smaller** — and decoded back to exactly the original, character for character. This is within the 20%–90% range quoted at the start of §9.7.

The codebook is what the theory predicts. The space character and the letter "a" — the two most common symbols in the sentence — got three-bit codewords; the letters "v" and "w", which occur once each, got six-bit codewords. The rare symbols sit deepest, exactly as Lemma 9.7's exchange argument demands.

The edge case bites. With only one distinct symbol, a naive implementation builds a tree consisting of a single leaf at depth 0, so the codeword is the empty string and the encoded file is zero bits long — from which nothing can be decoded. The program handles it by pairing the lone symbol with a dummy, giving it the one-bit codeword `0`, so 1000 characters cost 1000 bits and the round trip succeeds. Every real compressor needs this special case.

Part 7 timed the construction from 4 000 to 512 000 symbols. The times were 0.902, 1.821, 3.929, 10.200, 24.001, 65.980, 182.586 and 461.363 milliseconds. Dividing by $n\lg n$ gives 18.8,17.6,17.6,21.3,23.5,30.4,39.7,47.5 — a slowly rising column, not a flat one. The asymptotic bound of Theorem 9.10 is still correct; what the rising column shows is that this implementation allocates $2n-1$ separate nodes with `new` and chases pointers scattered across memory, so the *constant* degrades as the working set outgrows the caches. An implementation using one flat array of nodes and indices instead of pointers flattens the column out. It is a useful reminder that $O(n\lg n)$ is a statement about growth, not a promise about memory behavior.

9.8 Matroids and Greedy Methods

9.8.1 The question this section answers

We have now proved three greedy algorithms correct — activity selection, the fractional knapsack, Huffman — and each proof was a fresh piece of work. That is unsatisfying. A mathematician's instinct is to ask: **is there a single structure such that, whenever a problem has that structure, greed is automatically correct?**

There is, and it is called a **matroid**. We must be careful about how much it claims:

> **WHAT MATROID THEORY DOES AND DOES NOT COVER.** To say it plainly: matroid theory does NOT cover all cases in which a greedy method applies — in particular **the activity-selection problem and the Huffman-coding problem are not covered**. But it does cover many cases of practical interest, including the minimum-spanning-tree problem and the task-scheduling problem of §9.8.6. Matroids are a sufficient condition for greed, not a necessary one.

So this section is not a replacement for §9.4 and §9.7; it is an additional tool. When your problem *is* a matroid, you get the correctness proof for free, which is a very good bargain.

9.8.2 The definition, and what every word means

Figure 9.11 states the definition.

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 $\varnothing \in \mathcal{I}$).
3. Exchange property: if $A, B \in \mathcal{I}$ and $|A| < |B|$, then some $x \in B - A$ has $A \cup \{x\} \in \mathcal{I}$.

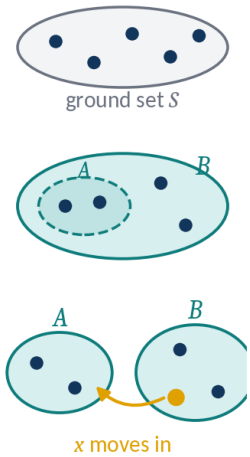


Figure 9.11 The definition of a matroid: a finite ground set S, a hereditary family $\mathcal{I}$ of *independent* subsets, and the exchange property.

That is three conditions on a pair $M = (S, \mathcal{I})$. Here they are again, each followed by a translation into ordinary language.

1. S is a finite set.

Translation: there is a bag of things. Edges of a graph, tasks to be scheduled, rows of a matrix — anything, as long as there are finitely many. S is called the **ground set**.

2. $\mathcal{I}$ is a non-empty family of subsets of S, called the independent subsets, such that if $B \in \mathcal{I}$ and $A \subseteq B$ then $A \in \mathcal{I}$. We say $\mathcal{I}$ is hereditary.

Translation: some collections of things are "OK" and the rest are "broken", and **throwing things away can never break an OK collection**. If a set of edges contains no cycle, then removing an edge still leaves no cycle. If a set of tasks can all be done on time, then dropping one still leaves a set that can all be done on time. Note an important consequence: since $\mathcal{I}$ is non-empty it contains some set B, and $\varnothing \subseteq B$, so **the empty set is always independent**. That makes sense: an empty collection cannot be broken.

The word *independent* is borrowed from linear algebra and it is the one that frightens people. It does not mean anything statistical. Read it as **"internally consistent"**, or simply **"legal"**.

3. If $A \in \mathcal{I}$, $B \in \mathcal{I}$, and $|A| < |B|$, then there exists some element $x \in B - A$ such that $A \cup \{x\} \in \mathcal{I}$. We say M satisfies the exchange property.

Translation: **if your legal collection is smaller than somebody else's, you can always steal one of their items and stay legal.** Not necessarily any item of theirs — *some* item. This is the condition that does all the work, and it is the one a random problem will fail.

> **IN PLAIN LANGUAGE.** A matroid is a rule for deciding which collections are legal, satisfying two fairness conditions. *First*, legality is not spoiled by giving something up. *Second*, nobody can be stuck: if your legal collection is smaller than another legal collection, there is always a way to grow yours by one, using something from theirs. That second condition is what stops a greedy algorithm painting itself into a corner — it guarantees that greed can always keep going for as long as anyone else could.

9.8.3 A worked example: the graphic matroid

Abstract definitions need an example immediately. **Figure 9.12** gives one.

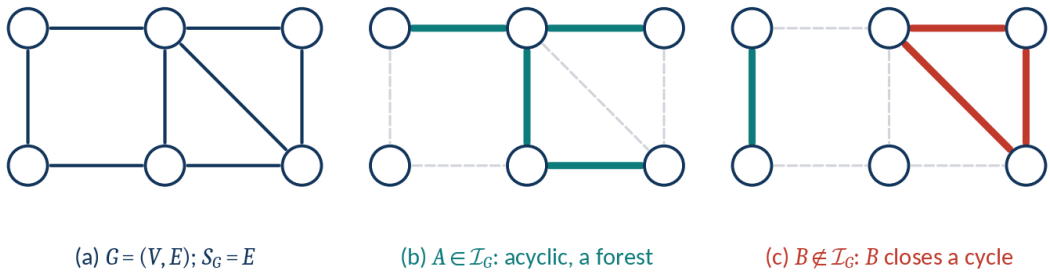


Figure 9.12 The graphic matroid $M_G = (S_G, \mathcal{I}_G)$ of an undirected graph $G = (V, E)$: the ground set is E, and a set of edges is independent exactly when it is acyclic — that is, when it forms a forest. **(a)** A graph G; **(b)** an independent set A of four edges with no cycle; **(c)** a 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.

Take an undirected graph $G = (V, E)$, where V is the set of vertices (dots) and E the set of edges (lines joining dots). Set

- $S_G = E$, the ground set is the edges;
- $A \in \mathcal{I}_G$ if and only if A is **acyclic** — that is, the subgraph $G_A = (V, A)$ contains no cycle, so it is a **forest** (a collection of trees).

Check the three conditions.

1. **Finite.** A graph has finitely many edges. ✓
2. **Hereditary.** If a set of edges contains no cycle, can removing an edge create one? No — a cycle in a subset would be a cycle in the whole. ✓ (And $\varnothing$ is acyclic. ✓)
3. **Exchange.** Suppose A and B are both acyclic and $|A| < |B|$. A forest on $|V|$ vertices with k edges has exactly $|V| - k$ connected components (each edge added to an empty graph joins two components into one). So G_A has $|V| - |A|$ components and G_B has $|V| - |B|$ components; since $|A| < |B|$, G_A has strictly **more** components than G_B. Now B, having fewer components, must contain an edge (u, v) whose endpoints u and v lie in *different* components of G_A — for if every edge of B joined two vertices inside one component of G_A, then every component of G_B would sit inside a component of G_A, giving G_B at least as many components as G_A. Adding that edge (u, v) to A joins two distinct components and therefore cannot create a cycle (a cycle needs a second route between u and v, and there is none across components). So $A \cup \{(u, v)\} \in \mathcal{I}_G$, and $(u, v) \notin A$ since it crosses components of G_A. ✓

So the graphic matroid really is a matroid, and the exchange property is the statement that a small forest can always be grown using an edge from a bigger forest. Chapter 11 will use exactly this fact to prove Kruskal's algorithm correct — you have now seen the proof, in advance, in a different language.

> **A NON-EXAMPLE, TO SEE WHAT FAILS.** Let S be the activities of **Figure 9.1** and let $\mathcal{I}$ be the sets of mutually compatible activities. Conditions 1 and 2 hold — dropping an activity cannot create an overlap. But condition 3 fails. Take $A = \{a_3\} = \{[0,6)\}$ and $B = \{a_1, a_4\} = \{[1,4), [5,7)\}$; both are compatible, and $|A| = 1 < 2 = |B|$. Can we add an element of B to A and stay compatible? $a_1 = [1,4)$ overlaps $[0,6)$; $a_4 = [5,7)$ also overlaps $[0,6)$. Neither works. The exchange property fails, and this is exactly why activity selection is **not** covered by matroid theory, even though greed solves it. Its correctness needed the bespoke argument of §9.4.7.

9.8.4 Two consequences of the exchange property

The exchange property has two immediate corollaries that explain *why* matroids make greed work.

LEMMA 9.11 (extension). Let $M = (S, \mathcal{I})$ be a matroid and $A \in \mathcal{I}$. If there exists any $B \in \mathcal{I}$ with $|B| > |A|$, then A can be extended: some $x \in S - A$ has $A \cup \{x\} \in \mathcal{I}$.

Proof. This is the exchange property restated; the extending element may be taken from $B - A$. ■

LEMMA 9.12. All **maximal** independent subsets of a matroid have the **same size**.

Proof. Call A maximal if no element can be added to it while staying independent. Suppose A and B are both maximal and $|A| < |B|$. By the exchange property there is $x \in B - A$ with $A \cup \{x\} \in \mathcal{I}$ — contradicting the maximality of A. So $|A| \geq |B|$; by symmetry $|B| \geq |A|$; hence $|A| = |B|$. ■

THIS IS THE WHOLE REASON MATROIDS WORK. Recall §9.4.2's distinction: *maximal* (cannot be extended) is easy, *maximum* (nothing is bigger) is hard. Lemma 9.12 says that **in a matroid the distinction collapses** — every maximal independent set is automatically of maximum size. A greedy algorithm produces a maximal set effortlessly, by construction. In a matroid, effortless is enough.

For the graphic matroid, Lemma 9.12 says every maximal forest has the same number of edges — which, for a connected graph, is the familiar statement that every spanning tree has exactly $|V| - 1$ edges.

9.8.5 Weighted matroids and the greedy algorithm

To have an optimization problem we need numbers.

DEFINITION. A matroid $M = (S, \mathcal{I})$ is **weighted** if it comes with a **weight function** w assigning a strictly positive weight $w(x) > 0$ to each element $x \in S$. The weight of a subset is $w(A) = \sum_{x \in A} w(x)$. An **optimal** subset of a weighted matroid is an independent subset of **maximum total weight**.

Since weights are positive, an optimal subset is always maximal — adding anything you could add would increase the weight — and therefore, by Lemma 9.12, of maximum size as well.

Pseudocode 9.1 gives the algorithm; its heart is three lines.

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
```

Its running time is easy to bound. Sorting $n = |S|$ elements (line 2) costs $O(n \lg n)$. The loop of lines 3–5 runs n times, and each iteration performs one independence test (line 4). If a test costs $f(n)$, the total is

$$O(n \lg n + n\, f(n)). \tag{9.11}$$

For the graphic matroid a test is a union-find operation, essentially $O(1)$, so Equation (9.11) becomes $O(n \lg n)$ — and the algorithm *is* Kruskal's. For the task-scheduling matroid below, a naive test costs $O(n)$, giving $O(n^2)$, which can be reduced to $O(n \lg n)$ with a disjoint-set structure.

Why is it correct? The two ingredients, as always.

LEMMA 9.13 (matroids have the greedy-choice property). Let $M = (S, \mathcal{I})$ be a weighted matroid with S sorted into monotonically decreasing order of weight. Let x be the **first** element of S such that $\{x\}$ is independent, if such an element exists. Then there is an optimal subset A of S that contains x.

Proof (exchange argument). Suppose B is a non-empty optimal subset. We build from it an optimal subset containing x.

1. If $x \in B$ we are done, so assume $x \notin B$.
2. No element of B has weight greater than $w(x)$. *Why:* every $y \in B$ has $\{y\} \subseteq B$ independent by heredity, so y is one of the elements for which the singleton is independent; and x was the *first* such element in decreasing weight order, so $w(y) \leq w(x)$.
3. Build a set A as follows. Start with $A_0 = \{x\}$, which is independent. While $|A_k| < |B|$, apply the exchange property to A_k and B to obtain an element of $B - A_k$ that can be added; let A_{k+1} be the result. This terminates with a set A satisfying $|A| = |B|$, $A \in \mathcal{I}$, and $A \subseteq \{x\} \cup B$ — in fact $A = \{x\} \cup B \setminus \{y\}$ for exactly one element $y \in B$, since A contains x plus $|B| - 1$ elements of B.
4. Now compare weights:

$$w(A) = w(B) - w(y) + w(x).$$

5. By step 2, $w(x) \geq w(y)$, so $w(A) \geq w(B)$.
6. But B was optimal, so $w(A) \leq w(B)$. Hence $w(A) = w(B)$ and A is optimal too — and $x \in A$. ■

LEMMA 9.14 (optimal substructure). Let x be the first element of S chosen by **Greedy** for the weighted matroid $M = (S, \mathcal{I})$. The remaining problem of finding a maximum-weight independent subset containing x reduces to finding a maximum-weight independent subset of the weighted matroid $M' = (S', \mathcal{I}')$, where

$$S' = \{\, y \in S \,:\, \{x, y\} \in \mathcal{I} \,\}, \qquad \mathcal{I}' = \{\, B \subseteq S' \,:\, B \cup \{x\} \in \mathcal{I} \,\},$$

with the same weights. (The pair M' is called the **contraction** of M by x, and it is again a matroid.)

Proof. If A is an independent subset of M containing x, then $A \setminus \{x\}$ is independent in M' by the definition of $\mathcal{I}'$, and $w(A) = w(A \setminus \{x\}) + w(x)$. Conversely if B is independent in M' then $B \cup \{x\}$ is independent in M, with $w(B \cup \{x\}) = w(B) + w(x)$. So the two problems' objective values differ by the constant $w(x)$, and a maximizer of one is a maximizer of the other. ■

THEOREM 9.15. Greedy(M, w) returns an optimal subset of the weighted matroid M.

Proof. By induction on $|S|$. If no singleton is independent, every independent set is empty and **Greedy** returns $\varnothing$, which is optimal. Otherwise let x be the first element for which $\{x\}$ is independent — the first element **Greedy** accepts. By Lemma 9.13 some optimal subset contains x, so taking x is safe. By Lemma 9.14 what remains is the same problem on the contraction M', whose ground set is smaller. The remaining iterations of **Greedy** are exactly a run of **Greedy** on M' with the elements still in decreasing weight order, so by the inductive hypothesis they return an optimal subset of M'. Adding x gives an optimal subset of M. ■

COMPARE THE EFFORT. In §9.4.7 we proved the greedy-choice property for one specific problem, and the proof was half a page. Lemma 9.13 proves it **once, for every weighted matroid there will ever be**. That is the payoff of abstraction: recognize your problem as a matroid and the correctness proof is already written.

9.8.6 The application: scheduling unit-time tasks with deadlines and penalties

DEFINITION. A **unit-time task** is a job that takes exactly one unit of time on a single processor. We are given n tasks $a_1, \dots, a_n$; task a_i has an integer **deadline** d_i with $1 \le d_i \le n$, and an integer **penalty** $w_i > 0$ which we must pay if the task finishes **after** its deadline. A **schedule** is a permutation of the tasks; the task in position j runs during $[j-1, j)$ and finishes at time j. A task is **late** if it finishes after its deadline, and **early** otherwise. The problem: find a schedule minimizing the total penalty of the late tasks.

Two simplifications make this tractable, and both are worth seeing.

Simplification 1: only the *set* of early tasks matters. Given any schedule, we can rearrange it into **early-first form**: put all the early tasks first, in non-decreasing order of deadline, then all the late tasks in any order. This does not make any early task late. *Why:* suppose early task a was in position j (so $j \le d_a$) and after rearrangement it is in position j'. In early-first form, the tasks in positions $1, \dots, j'$ — that is, a itself and every task placed before it — are all early tasks with deadline $\le d_a$, because the early tasks are sorted by deadline. Now count. All j' of these tasks were early in the original schedule too, so there each one finished no later than its own deadline, which is at most d_a. So in the original schedule these j' tasks occupied j' different slots, all numbered $\le d_a$. There are only d_a such slots, so $j' \le d_a$: task a finishes on time in its new position as well. So we may restrict attention to early-first schedules, and such a schedule is determined by which set of tasks is early.

Simplification 2: minimizing the penalty of the late tasks is the same as maximizing the penalty of the early tasks. The total of all penalties, $\sum_i w_i$, is a constant. So

$$\text{minimize} \sum_{\text{late}} w_i \Leftrightarrow \text{maximize} \sum_{\text{early}} w_i. \tag{9.12}$$

By Equation (9.12) we have turned a minimization into a maximization, which is what a weighted matroid wants.

DEFINITION. A set A of tasks is **independent** if there exists a schedule in which **no task of A is late**. For $t = 0,1, \dots, n$, let

$$N_t(A) = |\{\, a_i \in A : d_i \le t \,\}| \tag{9.13}$$

be the number of tasks in A whose deadline is t or earlier.

LEMMA 9.16. For any set A of unit-time tasks, the following three statements are equivalent. **(1)** A is independent. **(2)** $N_t(A) \le t$ for every $t = 0,1, \dots, n$. **(3)** If the tasks of A are scheduled in non-decreasing order of deadline, no task of A is late.

Proof. We prove (1) ⇒ (2) ⇒ (3) ⇒ (1).

(1) ⇒ (2). Suppose A is independent, so some schedule makes no task of A late. Fix t. Every task in A with deadline $\le t$ must finish by time t in that schedule. Only the time slots $1,2, \dots, t$ finish at or before time t, and each holds one unit-time task. So at most t tasks can finish by time t, giving $N_t(A) \le t$.

(2) ⇒ (3). Assume $N_t(A) \le t$ for all t, and schedule A in non-decreasing order of deadline. Let a task a occupy position j, so it finishes at time j. The j tasks in positions $1, \dots, j$ all have deadline $\le d_a$, because the order is non-decreasing. Hence there are at least j tasks in A with deadline $\le d_a$, that is, $N_{d_a}(A) \ge j$. By hypothesis $N_{d_a}(A) \le d_a$. Chaining: $j \le N_{d_a}(A) \le d_a$, so a finishes at time $j \le d_a$ and is not late.

(3) ⇒ (1). Statement (3) exhibits a schedule of A in which no task of A is late; extend it by appending the tasks not in A in any order. That is a schedule witnessing the independence of A. ■

Condition (2), with $N_t(A)$ counted as in Equation (9.13), is the *test*, and it is what the program implements: to check whether a set of tasks is independent, count for each t how many of them are due by time t, and verify the count never exceeds t.

THEOREM 9.17. Let S be the set of all n unit-time tasks and let $\mathcal{I}$ be the family of independent sets of tasks. Then $M = (S, \mathcal{I})$ is a matroid.

Proof. Three conditions.

1. S **is finite.** There are n tasks. ✓
2. $\mathcal{I}$ **is hereditary.** Let $B \in \mathcal{I}$ and $A \subseteq B$. For every t, $N_t(A) \le N_t(B)$ because A has no tasks that B lacks; and $N_t(B) \le t$ by Lemma 9.16(2). So $N_t(A) \le t$ for all t, and $A \in \mathcal{I}$. ✓ (And $\varnothing$ is independent, since $N_t(\varnothing) = 0 \le t$.)
3. **The exchange property.** Let $A, B \in \mathcal{I}$ with $|A| < |B|$. We must find $x \in B - A$ with $A \cup \{x\} \in \mathcal{I}$.
   - Since every deadline is at most n, $N_n(A) = |A|$ and $N_n(B) = |B|$, so $N_n(A) < N_n(B)$.
   - Also $N_0(A) = N_0(B) = 0$, since no deadline is ≤ 0.
   - Let k be the **largest** value of t in $\{0,1,\dots,n\}$ for which $N_t(A) \ge N_t(B)$. Such a k exists because $t = 0$ qualifies, and $k < n$ because $t = n$ does not. By maximality, $N_t(A) < N_t(B)$ for every $t > k$.
   - Consider the tasks with deadline **exactly** $k+1$. In A there are $N_{k+1}(A) - N_k(A)$ of them; in B there are $N_{k+1}(B) - N_k(B)$. Now $N_{k+1}(B) > N_{k+1}(A)$ (since $k+1 > k$) and $N_k(B) \le N_k(A)$ (by the choice of k), so

$$N_{k+1}(B) - N_k(B) \;>\; N_{k+1}(A) - N_k(A).$$

   - So B contains strictly more tasks of deadline exactly $k+1$ than A does. Therefore the set of B's tasks with deadline $k+1$ is not a subset of A's, and we may pick a task $x \in B$ with $d_x = k+1$ and $x \notin A$.
   - **Check** $A \cup \{x\}$ **is independent** using Lemma 9.16(2). For $t \le k$: x has deadline $k+1 > t$, so $N_t(A \cup \{x\}) = N_t(A) \le t$ ✓. For $t \ge k+1$: $N_t(A \cup \{x\}) = N_t(A) + 1 \le N_t(B) \le t$, where the first inequality uses $N_t(A) < N_t(B)$ for $t > k$ and the second is Lemma 9.16(2) for B ✓.
   - Hence $A \cup \{x\} \in \mathcal{I}$. ✓ ■

By Theorem 9.15, therefore, the greedy algorithm — **take the tasks in decreasing order of penalty, keeping each one if the set stays independent** — solves the scheduling problem optimally. No further proof is needed. That is the whole point of matroids.

9.8.7 Worked by hand on an example

Figure 9.13 gives the instance we shall work by hand.

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 $a_1, \dots, a_7$ with deadlines d_i and penalties w_i.

The data: deadlines $d = \langle 4,2,4,3,1,4,6 \rangle$ and penalties $w = \langle 70,60,50,40,30,20,10 \rangle$. Their total is

$$70 + 60 + 50 + 40 + 30 + 20 + 10 = 280,$$

which is what we would pay if every task were late. Conveniently, the tasks are already listed in decreasing order of penalty, so **Greedy** considers them as $a_1, a_2, \dots, a_7$.

Now run it, testing independence with Lemma 9.16(2) at each step.

Consider a_1 **(**$d = 4$**,** $w = 70$**).** Trial set $\{a_1\}$. Counts: $N_1 = 0$, $N_2 = 0$, $N_3 = 0$, $N_4 = 1$, $N_5 = 1$, $N_6 = 1$, $N_7 = 1$. Each $N_t \le t$? $0 \le 1$, $0 \le 2$, $0 \le 3$, $1 \le 4$, … ✓ **Keep.** $A = \{a_1\}$.

Consider a_2 **(**$d = 2$**,** $w = 60$**).** Trial set $\{a_1, a_2\}$. $N_1 = 0 \le 1$ ✓, $N_2 = 1 \le 2$ ✓, $N_3 = 1 \le 3$ ✓, $N_4 = 2 \le 4$ ✓, rest fine. **Keep.** $A = \{a_1, a_2\}$.

Consider a_3 **(**$d = 4$**,** $w = 50$**).** Trial set $\{a_1, a_2, a_3\}$. $N_1 = 0$, $N_2 = 1 \le 2$ ✓, $N_3 = 1 \le 3$ ✓, $N_4 = 3 \le 4$ ✓. **Keep.** $A = \{a_1, a_2, a_3\}$.

Consider a_4 **(**$d = 3$**,** $w = 40$**).** Trial set $\{a_1, a_2, a_3, a_4\}$. $N_1 = 0 \le 1$ ✓, $N_2 = 1 \le 2$ ✓, $N_3 = 2 \le 3$ ✓, $N_4 = 4 \le 4$ ✓ — exactly at the limit. **Keep.** $A = \{a_1, a_2, a_3, a_4\}$.

Consider a_5 **(**$d = 1$**,** $w = 30$**).** Trial set $\{a_1, a_2, a_3, a_4, a_5\}$. $N_1 = 1 \le 1$ ✓, $N_2 = 2 \le 2$ ✓, $N_3 = 3 \le 3$ ✓, $N_4 = \mathbf{5} > 4$ ✗. **Reject** — five tasks all due by time 4 cannot all be done in four slots.

Consider a_6 **(**$d = 4$**,** $w = 20$**).** Trial set $\{a_1, a_2, a_3, a_4, a_6\}$. $N_4 = 5 > 4$ ✗. **Reject**, for the same reason.

Consider a_7 **(**$d = 6$**,** $w = 10$**).** Trial set $\{a_1, a_2, a_3, a_4, a_7\}$. $N_4 = 4 \le 4$ ✓, $N_5 = 4 \le 5$ ✓, $N_6 = 5 \le 6$ ✓, $N_7 = 5 \le 7$ ✓. **Keep.** $A = \{a_1, a_2, a_3, a_4, a_7\}$.

So the early set is $A = \{a_1, a_2, a_3, a_4, a_7\}$, of weight

$$70 + 60 + 50 + 40 + 10 = 230,$$

and the late set is $\{a_5, a_6\}$, of weight

$$30 + 20 = 50.$$

Check: $230 + 50 = 280$ ✓. The minimum total penalty is **50**.

To produce the actual schedule, put the early tasks in non-decreasing deadline order (Lemma 9.16(3)) and append the late ones: $\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle$. Verify slot by slot: a_2 in slot 1 with

$d = 2 \geq 1$ ✓; a_4 in slot 2 with $d = 3 \geq 2$ ✓; a_1 in slot 3 with $d = 4 \geq 3$ ✓; a_3 in slot 4 with $d = 4 \geq 4$ ✓; a_7 in slot 5 with $d = 6 \geq 5$ ✓; a_5 in slot 6 with $d = 1 < 6$ — late, as expected; a_6 in slot 7 with $d = 4 < 7$ — late. **Table 9.10** records the run.

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	$N_4 = 1 \leq 4$	keep	$\{a_1\}$
a_2	2	60	$N_2 = 1 \leq 2, N_4 = 2 \leq 4$	keep	$\{a_1, a_2\}$
a_3	4	50	$N_4 = 3 \leq 4$	keep	$\{a_1, a_2, a_3\}$
a_4	3	40	$N_3 = 2 \leq 3, N_4 = 4 \leq 4$	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	$N_6 = 5 \leq 6$	keep	$\{a_1, a_2, a_3, a_4, a_7\}$

The program in **Listing 9.5** confirms it, and checks it against brute force over all $2^7 = 128$ subsets:

```
early (on time) set A = {a1, a2, a3, a4, a7},  weight saved = 230
late set              = {a5, a6},  penalty paid = 50
schedule              : a2(slot 1, d=2, on time)
                        a4(slot 2, d=3, on time)
                        a1(slot 3, d=4, on time)
                        a3(slot 4, d=4, on time)
                        a7(slot 5, d=6, on time)
                        a5(slot 6, d=1, LATE)
                        a6(slot 7, d=4, LATE)

brute force over all 2^7 = 128 subsets:
  best independent set = {a1, a2, a3, a4, a7},  weight saved = 230,  minimum penalty = 50
  greedy matches the optimum: YES

stress test, 200000 random 8-task instances:
  greedy differed from the optimum in 0 of 200000 instances
```

Two hundred thousand random instances, and greedy matched the exhaustive optimum on every one. It was never going to do otherwise: Theorem 9.17 says the structure is a matroid, and Theorem 9.15 says greed is then optimal. The stress test is not evidence for the theorem; it is a check that the *program* implements the theorem.

9.8.8 One algorithm, two problems

Listing 9.5 writes **Greedy**(M, w) exactly once, against an abstract independence oracle, and then feeds it two different oracles. With the task-scheduling oracle it solves the problem above. With an "is this set of edges acyclic?" oracle it becomes **Kruskal's minimum-spanning-tree algorithm** — the subject of Chapter 11 — without a single line of the greedy loop being changed.

One wrinkle: **Greedy** *maximizes* weight, and a minimum spanning tree *minimizes* it. The fix is a standard trick: run the algorithm on the transformed weights $w'(e) = w_{\max} + 1 - w(e)$, which reverses the order of the edges without changing which sets are independent. Maximizing w' over a maximal forest is the same as minimizing w over it, because a spanning tree of a connected graph always has exactly $|V| - 1$ edges (Lemma 9.12 again), so $\sum w'(e) = (|V| - 1)(w_{\max} + 1) - \sum w(e)$ and the first term is a constant.

Listing 9.5 **Greedy**(M, w) written once against an abstract independence oracle, then applied to the task-scheduling matroid (with a brute-force check and a 200 000-instance stress test) and to the graphic matroid, where it becomes Kruskal's algorithm. Compile with `g++ -std=c++17 -O2 -o matroid matroid.cpp` and run `./matroid`.

```
// matroid.cpp — the weighted-matroid greedy algorithm, once, used twice.
//   * GREEDY(M, w) is written ONCE against an abstract independence oracle;
//   * instance 1: scheduling unit-time tasks with deadlines and penalties, on
//     the seven tasks of Section 9.8.7, checked against brute force;
//   * instance 2: the graphic matroid, which turns the same code into Kruskal's
//     minimum-spanning-tree algorithm (Chapter 11).
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <functional>
#include <algorithm>
#include <numeric>
#include <random>
using namespace std;

// ------------------------------------------------------------ GREEDY(M,w) --
// S is the ground set, given as indices 0..n-1.  w[i] is the weight of element i.
// independent(A) decides whether the subset A belongs to the family I.
// The algorithm sorts S into monotonically DECREASING order by weight and adds
// each element if doing so keeps the set independent.
vector<int> matroidGreedy(int n, const vector<long long>& w,
                          const function<bool(const vector<int>&)>& independent,
                          bool trace, const vector<string>& name) {
    vector<int> order(n);
    iota(order.begin(), order.end(), 0);
    sort(order.begin(), order.end(), [&](int a, int b) {
        if (w[a] != w[b]) return w[a] > w[b];
        return a < b;                                  // reproducible tie-break
    });
    vector<int> A;
    for (int x : order) {
        A.push_back(x);
        bool ok = independent(A);
        if (!ok) A.pop_back();
        if (trace) {
            cout << "    consider " << setw(4) << name[x] << " (weight " << setw(3) << w[x]
<< "): "
                 << (ok ? "keep   A = {" : "REJECT A = {");
            for (size_t i = 0; i < A.size(); ++i) { if (i) cout << ", "; cout <<
name[A[i]]; }
            cout << "}\n";
        }
    }
    return A;
}
```

Listing 9.5 (Continued) *Greedy(M,w) written once against an abstract independence oracle, then applied to the task-scheduling matroid (with a brute-force check and a 200 000-instance stress test) and to the graphic matroid, where it becomes Kruskal's algorithm. Compile with g++ -std=c++17 -O2 -o matroid matroid.cpp and run ./matroid.*

```
// ------------------------------ instance 1: unit-time task scheduling -------
// A set A of tasks is independent iff there is a schedule in which no task of A
// is late.  The test: for every t, the number of tasks in A with deadline <= t
// must be at most t.
static vector<int> DEADLINE;

bool taskIndependent(const vector<int>& A) {
    int n = (int)DEADLINE.size();
    for (int t = 1; t <= n; ++t) {
        int cnt = 0;
        for (int i : A) if (DEADLINE[i] <= t) ++cnt;
        if (cnt > t) return false;
    }
    return true;
}

int main() {
    cout << "=== Instance 1: the seven unit-time tasks of Section 9.8.7 ===\n";
    vector<string> tname = { "a1", "a2", "a3", "a4", "a5", "a6", "a7" };
    DEADLINE =            {    4,    2,    4,    3,    1,    4,    6  };
    vector<long long> pen = {  70,   60,   50,   40,   30,   20,   10  };
    int n = 7;

    cout << "  task     :"; for (auto& s : tname) cout << setw(6) << s;   cout << "\n";
    cout << "  deadline :"; for (int d : DEADLINE) cout << setw(6) << d;  cout << "\n";
    cout << "  penalty  :"; for (long long p : pen) cout << setw(6) << p; cout << "\n";
    long long totalPen = accumulate(pen.begin(), pen.end(), 0LL);
    cout << "  total penalty if every task were late = " << totalPen << "\n\n";

    cout << "  GREEDY, taking tasks in decreasing order of penalty:\n";
    vector<int> A = matroidGreedy(n, pen, taskIndependent, true, tname);

    cout << "\n  early (on time) set A = {";
    vector<int> sortedA = A; sort(sortedA.begin(), sortedA.end());
    for (size_t i = 0; i < sortedA.size(); ++i) { if (i) cout << ", "; cout <<
tname[sortedA[i]]; }
    long long saved = 0; for (int i : A) saved += pen[i];
    cout << "},  weight saved = " << saved << "\n";

    cout << "  late set             = {";
    vector<int> late; long long paid = 0;
    for (int i = 0; i < n; ++i)
        if (find(A.begin(), A.end(), i) == A.end()) { late.push_back(i); paid += pen[i]; }
    for (size_t i = 0; i < late.size(); ++i) { if (i) cout << ", "; cout << tname[late[i]];
}
    cout << "},  penalty paid = " << paid << "\n";
```

Listing 9.5 (Continued) *Greedy(M,w) written once against an abstract independence oracle, then applied to the task-scheduling matroid (with a brute-force check and a 200 000-instance stress test) and to the graphic matroid, where it becomes Kruskal's algorithm. Compile with g++ -std=c++17 -O2 -o matroid matroid.cpp and run ./matroid.*

```
    // the actual schedule: early tasks in order of deadline, then the late ones
    vector<int> sched = A;
    sort(sched.begin(), sched.end(), [&](int a, int b) {
        if (DEADLINE[a] != DEADLINE[b]) return DEADLINE[a] < DEADLINE[b];
        return a < b;
    });
    for (int i : late) sched.push_back(i);
    cout << "  schedule             : ";
    for (size_t i = 0; i < sched.size(); ++i) {
        cout << tname[sched[i]] << "(slot " << i + 1 << ", d=" << DEADLINE[sched[i]]
             << (DEADLINE[sched[i]] >= (int)i + 1 ? ", on time" : ", LATE") << ")";
        if (i + 1 < sched.size()) cout << "\n                          ";
    }
    cout << "\n\n";

    cout << "  brute force over all 2^7 = 128 subsets:\n";
    long long bestSaved = -1; int bestMask = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        vector<int> B;
        for (int i = 0; i < n; ++i) if (mask >> i & 1) B.push_back(i);
        if (!taskIndependent(B)) continue;
        long long s = 0; for (int i : B) s += pen[i];
        if (s > bestSaved) { bestSaved = s; bestMask = mask; }
    }
    cout << "    best independent set = {";
    bool first = true;
    for (int i = 0; i < n; ++i) if (bestMask >> i & 1) { if (!first) cout << ", "; cout <<
tname[i]; first = false; }
    cout << "},  weight saved = " << bestSaved
         << ",  minimum penalty = " << totalPen - bestSaved << "\n";
    cout << "    greedy matches the optimum: "
         << (bestSaved == saved ? "YES" : "NO") << "\n\n";

    cout << "  stress test, 200000 random 8-task instances:\n";
    mt19937 rng(20260919);
    int mismatches = 0;
    for (int t = 0; t < 200000; ++t) {
        int m = 8;
        DEADLINE.assign(m, 0);
        vector<long long> w(m);
        for (int i = 0; i < m; ++i) { DEADLINE[i] = 1 + (int)(rng() % m); w[i] = 1 + (long
long)(rng() % 100); }
        vector<string> nm(m, "x");
        vector<int> G = matroidGreedy(m, w, taskIndependent, false, nm);
        long long gs = 0; for (int i : G) gs += w[i];
        long long bs = -1;
        for (int mask = 0; mask < (1 << m); ++mask) {
            vector<int> B;
```

Listing 9.5 (Continued) *Greedy(M,w) written once against an abstract independence oracle, then applied to the task-scheduling matroid (with a brute-force check and a 200 000-instance stress test) and to the graphic matroid, where it becomes Kruskal's algorithm. Compile with g++ -std=c++17 -O2 -o matroid matroid.cpp and run ./matroid.*

```
            for (int i = 0; i < m; ++i) if (mask >> i & 1) B.push_back(i);
            if (!taskIndependent(B)) continue;
            long long s = 0; for (int i : B) s += w[i];
            bs = max(bs, s);
        }
        if (gs != bs) ++mismatches;
    }
    cout << "    greedy differed from the optimum in " << mismatches << " of 200000
instances\n\n";

    // ------------------------- instance 2: the graphic matroid ---------------
    cout << "=== Instance 2: the same GREEDY on a graphic matroid = Kruskal ===\n";
    struct Edge { int u, v; long long w; string nm; };
    vector<Edge> E = {
        {0,1,4,"a-b"}, {0,7,8,"a-h"}, {1,2,8,"b-c"}, {1,7,11,"b-h"}, {2,3,7,"c-d"},
        {2,8,2,"c-i"}, {2,5,4,"c-f"}, {3,4,9,"d-e"}, {3,5,14,"d-f"}, {4,5,10,"e-f"},
        {5,6,2,"f-g"}, {6,7,1,"g-h"}, {6,8,6,"g-i"}, {7,8,7,"h-i"}
    };
    int V = 9;
    // independence: a set of edges is independent iff it is acyclic (a forest)
    auto acyclic = [&](const vector<int>& sel) {
        vector<int> parent(V); iota(parent.begin(), parent.end(), 0);
        function<int(int)> find_ = [&](int x) { while (parent[x] != x) x = parent[x] =
parent[parent[x]]; return x; };
        for (int i : sel) {
            int a = find_(E[i].u), b = find_(E[i].v);
            if (a == b) return false;                // this edge closes a cycle
            parent[a] = b;
        }
        return true;
    };
    // GREEDY maximizes weight, so to get a MINIMUM spanning tree we feed it
    // w'(e) = Wmax + 1 - w(e), which reverses the order without changing the matroid
    long long Wmax = 0; for (auto& e : E) Wmax = max(Wmax, e.w);
    vector<long long> wp(E.size());
    vector<string> enm(E.size());
    for (size_t i = 0; i < E.size(); ++i) { wp[i] = Wmax + 1 - E[i].w; enm[i] = E[i].nm; }
    vector<int> T = matroidGreedy((int)E.size(), wp, acyclic, false, enm);
    long long treeWeight = 0; for (int i : T) treeWeight += E[i].w;
    sort(T.begin(), T.end(), [&](int a, int b) { return E[a].w < E[b].w; });
    cout << "  minimum spanning tree edges (in the order the weights fall):\n   ";
    for (int i : T) cout << " " << E[i].nm << "(" << E[i].w << ")";
    cout << "\n  total weight = " << treeWeight << ",  edges chosen = " << T.size()
         << " (a spanning tree of " << V << " vertices needs " << V - 1 << ")\n";
    return 0;
}
```

The second instance printed:

```
=== Instance 2: the same GREEDY on a graphic matroid = Kruskal ===
  minimum spanning tree edges (in the order the weights fall):
    g-h(1) c-i(2) f-g(2) a-b(4) c-f(4) c-d(7) a-h(8) d-e(9)
  total weight = 37,  edges chosen = 8 (a spanning tree of 9 vertices needs 8)
```

Eight edges for nine vertices, total weight 37. You will meet this graph again in Chapter 11, where the same answer is obtained by Kruskal's algorithm written in its familiar form — and where the proof of correctness is, at bottom, Theorem 9.15 for the graphic matroid.

9.9 A Catalog of Greedy Algorithms Still to Come

The greedy method has many advanced applications, and it is worth setting them out now so that you recognize them when they arrive. **Table 9.11** is the map.

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

Three of these deserve a word now.

Kruskal's algorithm (Chapter 11) sorts the edges by weight and adds each one that does not create a cycle. §9.8.8 has already shown you that this is literally **Greedy**(M, w) on the graphic matroid, so its correctness is Theorem 9.15 and needs no new argument.

Prim's algorithm (Chapter 11) grows a single tree, repeatedly adding the lightest edge that leaves it. This is *not* the matroid greedy algorithm — it does not process edges in global weight order — and it needs its own proof. That proof is an exchange argument: given any minimum spanning tree, if it does not contain the lightest edge crossing the current cut, add that edge (creating a cycle), delete a heavier cycle edge crossing the same cut, and observe that the weight did not increase. Compare that with step 5 of Theorem 9.2 and you will see the same four moves.

Dijkstra's algorithm (Chapter 13) repeatedly finalizes the unvisited vertex with the smallest tentative distance. Again an exchange argument: if a shorter path to that vertex existed, it would have to leave the finalized region at some point, and the vertex where it leaves would have had a smaller tentative distance — contradiction. Note that this argument uses **non-negativity of the edge weights** at exactly one point, which is why Dijkstra fails on negative edges and Bellman-Ford (a dynamic program) is needed instead. That is the greedy-versus-dynamic-programming decision of §9.6, appearing again in a new costume.

9.10 Profitina in Practice

Profitina is an AI-visibility platform — a **GEO** (Generative Engine Optimization) product, live at `app.profitina.com` — which measures how ready a brand's website is to be *cited by AI answer engines*, from real signals such as content extractability, structured data and technical authority, and then helps fix what it finds. Three parts of that product are greedy problems of exactly the kind this chapter has been about. We describe the *shape* of each problem and the *class* of solution; the scoring internals are proprietary and are not the point here.

1. The re-scan scheduler is an activity-selection problem. Every brand on the platform is re-scanned automatically at a cadence that depends on its subscription tier. A scan occupies a worker for a bounded interval, and a worker can run only one scan at a time — precisely the "exclusive use of a common resource" of §9.4.1. On a busy night the scans requested exceed the workers available, so something must be dropped. Sorting the pending scans by the time each would *finish* and sweeping greedily maximizes the number of brands refreshed, at $\Theta(n\lg n)$, with the proof of Theorem 9.4 behind it. Sorting by the time each would *start* — the naive "first come, first served" — is the earliest-start rule of §9.4.3, and it is provably worse.

2. The crawl budget within one scan is a knapsack problem. A single scan has a page budget. Each candidate page costs some fetch-and-parse time and contributes some amount of diagnostic value. Pages cannot be half-fetched, so this is the **0-1** knapsack of §9.6.4 and greed is *wrong* — which is exactly the trap: ranking pages by value-per-millisecond and filling the budget gives an answer that is usually within a couple of percent and occasionally far worse, as **Table 9.6** and the stress test of §9.6.5 showed. The honest engineering choices are a dynamic program when the budget is small, or the greedy answer *plus* the fractional bound of Theorem 9.5 as an explicit measure of how much value might have been missed.

3. The historical analytics store is a compression problem. The dashboard keeps a weekly score trend per brand and offers CSV export, so the platform accumulates long runs of highly repetitive status and event codes. That is a skewed frequency distribution, which is where Huffman coding earns the 20%–90% savings quoted in §9.7. §9.7 is the whole method, and the round trip verified in **Listing 9.4** is the property that matters: a compressed log that cannot be decoded exactly is worthless.

> **THE PRODUCT ADVISES; THE USER DECIDES.** A scheduler that drops scans, and a crawl budget that skips pages, both change what a brand owner sees. Neither may be presented as a guarantee of completeness. And any scan of a site that touches personal data must respect UU PDP 27/2022 — which in scheduling terms means that a data-subject request has a **deadline**, and missing it carries a **penalty**: the task-scheduling matroid of §9.8.6, with the penalties being legal rather than monetary.

9.10.1 The worked case study: Profitina Laundry

The runnable example is always **Profitina Laundry**, a two-machine shop in Surabaya. **Listing 9.6** works one day at the shop, and the day contains all three problems.

Part A — the dryer. The shop has one industrial dryer and eleven booking requests. Which bookings should it accept? This is activity selection exactly. The program runs the proved-optimal rule and two rules the owner might plausibly prefer:

```
  earliest finish (greedy, proved optimal) -> 4 bookings served
      Pak Anto 07:30-09:00, Warung Bu Tin 09:30-11:30, Salon Ayu 12:00-14:00, Klinik Sehat
14:00-16:00
  first come, first served                 -> 3 bookings served
      Hotel Melati 07:00-10:00, Bu Wati 10:00-12:30, Pak Hendra 12:30-14:30
  longest booking first                    -> 3 bookings served
      Bu Sri 08:00-11:00, Kos Merdeka 11:00-13:30, Klinik Sehat 14:00-16:00

  brute force over all 2^11 = 2048 subsets: the maximum is 4 bookings
```

Four customers instead of three — a 33% increase in the number served, from changing one line of a sort comparator. "First come, first served" feels fair and loses a customer every day. "Longest booking first" feels profitable — longer loads, bigger bills — and also loses a customer.

Part B — the status log. The dryer emits one status character per minute: `R` running, `I` idle, `W` waiting for the operator, `E` error, `C` cool-down, `S` spin, `D` drain. Over a day that is 9 000 characters, and the shop's uplink is slow. Huffman-code it:

```
  9000 status characters, 7 distinct codes
    code       count    codeword      bits count x bits
       R        7580           1         1          7580
       S         356         011         3          1068
       C         355         001         3          1065
       I         355         010         3          1065
       D         118       00010         5           590
       E         118       00011         5           590
       W         118        0000         4           472
  B(T) = 12430 bits
  plain 8-bit log      : 72000 bits
  Huffman-coded log    : 12430 bits
  saving               : 82.74%
  round trip           : IDENTICAL
```

An 82.7% saving, and the log decodes back byte-identical. Notice why the saving is so large: `R` accounts for 7580 of the 9000 characters, and it got a **one-bit** codeword. The distribution is extremely skewed, and skew is what Huffman converts into compression. Notice too that `W`, `D` and `E` all occur 118 times yet `W` got four bits while `D` and `E` got five — the tie between equal frequencies was broken by the order of the merges, and any of those trees is equally optimal.

Part C — the delivery van. Finished laundry goes out in one van with a 60 kg limit. Five parcels are waiting, each with a delivery fee. Parcels cannot be split. This is the 0-1 knapsack, so the density rule is *wrong*:

```
  greedy by fee per kg : P1 + P2 + P4   = 615
  true optimum (DP)    : P1 + P2 + P5   = 683
  greedy leaves 68 on the table (10.0% of the optimum)
  if the parcels were bulk detergent that CAN be split, greedy would
  be optimal and would earn 758.00
```

Greedy takes P4 because it ties with P3 on fee per kilogram at 11.0 and is lighter; the optimum instead takes P5, whose density is the *worst* of the five at 8.0, because P5 weighs exactly the 25 kg that P1 and P2 leave free. Greedy loses 10% of the day's delivery revenue. And the last line is the lesson of §9.6 in one number: the same five loads, made divisible, would earn 758 under the same greedy rule — and that 758 is a legitimate upper bound on the indivisible answer of 683.

Listing 9.6 Profitina Laundry: activity selection on the dryer bookings (with a brute-force check), Huffman compression of the day's status log with a verified round trip, and the delivery van as a 0-1 knapsack where greed loses 10% of the revenue. Compile with `g++ -std=c++17 -O2 -o laundry laundry.cpp` and run `./laundry`.

```
// laundry.cpp — Profitina Laundry: three greedy decisions in one working day.
//   Part A  one industrial dryer, many booking requests: which bookings to take
//           (activity selection -- greedy is provably optimal).
//   Part B  the day's machine-status log, Huffman-compressed before it is sent
//           over the shop's slow uplink.
//   Part C  the delivery van, with whole parcels that cannot be cut in half
//           (0-1 knapsack -- greedy is NOT optimal, and the program shows it).
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <queue>
#include <map>
#include <algorithm>
#include <cstdint>
using namespace std;

static string hhmm(int minutesFrom7am) {
    int t = 7 * 60 + minutesFrom7am;
    char buf[16];
    snprintf(buf, sizeof buf, "%02d:%02d", t / 60, t % 60);
    return string(buf);
}

// ------------------------------------------------------- Part A: the dryer ---
struct Booking { string customer; int s, f; };   // minutes after 07:00

vector<int> earliestFinish(vector<Booking> b) {
    int n = (int)b.size();
    vector<int> idx(n);
    for (int i = 0; i < n; ++i) idx[i] = i;
    sort(idx.begin(), idx.end(), [&](int p, int q) {
        if (b[p].f != b[q].f) return b[p].f < b[q].f;
        return p < q;
    });
    vector<int> chosen;
    int freeAt = -1;
    for (int i : idx)
        if (b[i].s >= freeAt) { chosen.push_back(i); freeAt = b[i].f; }
    return chosen;
}
```

Listing 9.6 (Continued) *Profitina Laundry: activity selection on the dryer bookings (with a brute-force check), Huffman compression of the day's status log with a verified round trip, and the delivery van as a 0-1 knapsack where greed loses 10% of the revenue. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp and run ./laundry.*

```
vector<int> firstComeFirstServed(vector<Booking> b) {
    int n = (int)b.size();
    vector<int> idx(n);
    for (int i = 0; i < n; ++i) idx[i] = i;
    sort(idx.begin(), idx.end(), [&](int p, int q) { return b[p].s < b[q].s; });
    vector<int> chosen; int freeAt = -1;
    for (int i : idx) if (b[i].s >= freeAt) { chosen.push_back(i); freeAt = b[i].f; }
    return chosen;
}

vector<int> longestFirst(vector<Booking> b) {
    int n = (int)b.size();
    vector<int> idx(n);
    for (int i = 0; i < n; ++i) idx[i] = i;
    sort(idx.begin(), idx.end(), [&](int p, int q) {
        return (b[p].f - b[p].s) > (b[q].f - b[q].s);
    });
    vector<int> chosen;
    for (int i : idx) {
        bool clash = false;
        for (int j : chosen) if (!(b[i].s >= b[j].f || b[j].s >= b[i].f)) clash = true;
        if (!clash) chosen.push_back(i);
    }
    return chosen;
}

int bruteForceDryer(const vector<Booking>& b) {
    int n = (int)b.size(), best = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        vector<int> c;
        for (int i = 0; i < n; ++i) if (mask >> i & 1) c.push_back(i);
        bool ok = true;
        for (size_t i = 0; i + 1 < c.size() && ok; ++i)
            for (size_t j = i + 1; j < c.size() && ok; ++j)
                if (!(b[c[i]].s >= b[c[j]].f || b[c[j]].s >= b[c[i]].f)) ok = false;
        if (ok) best = max(best, (int)c.size());
    }
    return best;
}
```

Listing 9.6 (Continued) *Profitina Laundry: activity selection on the dryer bookings (with a brute-force check), Huffman compression of the day's status log with a verified round trip, and the delivery van as a 0-1 knapsack where greed loses 10% of the revenue. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp and run ./laundry.*

```
// ---------------------------------------------------- Part B: the log file ---
struct Node {
    long long freq; char ch; bool leaf; int serial; Node *l, *r;
    Node(long long f, char c) : freq(f), ch(c), leaf(true),  serial(0), l(nullptr),
r(nullptr) {}
    Node(Node* a, Node* b)    : freq(a->freq + b->freq), ch(0), leaf(false), serial(0),
l(a), r(b) {}
};
struct Later {
    bool operator()(const Node* a, const Node* b) const {
        if (a->freq != b->freq) return a->freq > b->freq;
        return a->serial > b->serial;
    }
};
Node* build(const map<char, long long>& f) {
    priority_queue<Node*, vector<Node*>, Later> Q;
    int serial = 0;
    vector<pair<char, long long>> v(f.begin(), f.end());
    sort(v.begin(), v.end(), [](const pair<char,long long>& a, const pair<char,long long>&
b){ return a.second < b.second; });
    for (auto& p : v) { Node* n = new Node(p.second, p.first); n->serial = serial++;
Q.push(n); }
    if (Q.size() == 1) { Node* o = Q.top(); Q.pop(); Node* z = new Node(o, new
Node(0,'\0')); return z; }
    while (Q.size() > 1) {
        Node* x = Q.top(); Q.pop();
        Node* y = Q.top(); Q.pop();
        Node* z = new Node(x, y); z->serial = serial++; Q.push(z);
    }
    return Q.top();
}
void codes(const Node* n, string c, map<char, string>& out) {
    if (!n) return;
    if (n->leaf) { out[n->ch] = c.empty() ? "0" : c; return; }
    codes(n->l, c + "0", out); codes(n->r, c + "1", out);
}

// ---------------------------------------------------- Part C: the van -----
struct Parcel { string id; int kg, fee; };

int vanGreedy(vector<Parcel> p, int cap, vector<string>& picked) {
    sort(p.begin(), p.end(), [](const Parcel& a, const Parcel& b) {
        return (double)a.fee / a.kg > (double)b.fee / b.kg;
    });
    int left = cap, got = 0;
    for (auto& x : p) if (x.kg <= left) { left -= x.kg; got += x.fee;
picked.push_back(x.id); }
    return got;
}
```

Listing 9.6 (Continued) *Profitina Laundry: activity selection on the dryer bookings (with a brute-force check), Huffman compression of the day's status log with a verified round trip, and the delivery van as a 0-1 knapsack where greed loses 10% of the revenue. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp and run ./laundry.*

```
int vanOptimal(const vector<Parcel>& p, int cap, vector<string>& picked) {
    int n = (int)p.size();
    vector<vector<int>> best(n + 1, vector<int>(cap + 1, 0));
    for (int i = 1; i <= n; ++i)
        for (int c = 0; c <= cap; ++c) {
            best[i][c] = best[i - 1][c];
            if (p[i - 1].kg <= c)
                best[i][c] = max(best[i][c], best[i - 1][c - p[i - 1].kg] + p[i - 1].fee);
        }
    int c = cap;
    for (int i = n; i >= 1; --i)
        if (best[i][c] != best[i - 1][c]) { picked.push_back(p[i - 1].id); c -= p[i -
1].kg; }
    reverse(picked.begin(), picked.end());
    return best[n][cap];
}

double vanFractional(vector<Parcel> p, int cap) {
    sort(p.begin(), p.end(), [](const Parcel& a, const Parcel& b) {
        return (double)a.fee / a.kg > (double)b.fee / b.kg;
    });
    double got = 0; int left = cap;
    for (auto& x : p) {
        if (left <= 0) break;
        if (x.kg <= left) { got += x.fee; left -= x.kg; }
        else { got += (double)x.fee * left / x.kg; left = 0; }
    }
    return got;
}

int main() {
    cout << "======== PART A: one industrial dryer, eleven booking requests ========\n";
    vector<Booking> b = {
        {"Bu Sri",    60, 240}, {"Pak Anto",  30, 120}, {"Hotel Melati", 0, 180},
        {"Warung Bu Tin", 150, 270}, {"Pak Yudi", 120, 300}, {"Bu Wati", 180, 330},
        {"Kos Merdeka", 240, 390}, {"Salon Ayu", 300, 420}, {"Pak Hendra", 330, 450},
        {"Bu Lastri", 390, 480}, {"Klinik Sehat", 420, 540}
    };
    int n = (int)b.size();
    cout << setw(16) << "customer" << setw(10) << "start" << setw(10) << "finish"
         << setw(12) << "minutes\n";
    for (auto& x : b)
        cout << setw(16) << x.customer << setw(10) << hhmm(x.s) << setw(10) << hhmm(x.f)
             << setw(12) << x.f - x.s << "\n";
```

Listing 9.6 (Continued) *Profitina Laundry: activity selection on the dryer bookings (with a brute-force check), Huffman compression of the day's status log with a verified round trip, and the delivery van as a 0-1 knapsack where greed loses 10% of the revenue. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp and run ./laundry.*

```
    struct { const char* name; vector<int> (*fn)(vector<Booking>); } rules[] = {
        { "earliest finish (greedy, proved optimal)", earliestFinish },
        { "first come, first served                ", firstComeFirstServed },
        { "longest booking first                   ", longestFirst },
    };
    cout << "\n";
    for (auto& R : rules) {
        vector<int> c = R.fn(b);
        sort(c.begin(), c.end(), [&](int p, int q) { return b[p].f < b[q].f; });
        cout << "  " << R.name << " -> " << c.size() << " bookings served\n      ";
        for (size_t i = 0; i < c.size(); ++i) {
            cout << b[c[i]].customer << " " << hhmm(b[c[i]].s) << "-" << hhmm(b[c[i]].f);
            if (i + 1 < c.size()) cout << ", ";
        }
        cout << "\n";
    }
    cout << "\n  brute force over all 2^" << n << " = " << (1 << n)
         << " subsets: the maximum is " << bruteForceDryer(b) << " bookings\n";

    cout << "\n======== PART B: compressing the day's status log ========\n";
    // One character per machine-minute. R = running, I = idle, W = waiting for
    // the operator, E = error, C = cool-down, S = spin, D = drain.
    string log;
    const char* pattern =
"RRRRRRRRSRRRRRRRRCIRRRRRRRRSRRRRRRRRCIRRRRRRRRDRRRRRRRRSWRRRRRRRRCIRRRRRRRRE";
    while (log.size() < 9000) log += pattern;
    log.resize(9000);
    map<char, long long> freq;
    for (char c : log) ++freq[c];
    Node* root = build(freq);
    map<char, string> code; codes(root, "", code);

    cout << "  " << log.size() << " status characters, " << freq.size() << " distinct
codes\n";
    cout << setw(8) << "code" << setw(12) << "count" << setw(12) << "codeword"
         << setw(10) << "bits" << setw(14) << "count x bits\n";
    long long bits = 0;
    vector<pair<char,long long>> fv(freq.begin(), freq.end());
    sort(fv.begin(), fv.end(), [](const pair<char,long long>&a, const pair<char,long
long>&b){ return a.second > b.second; });
    for (auto& p : fv) {
        int d = (int)code[p.first].size();
        cout << setw(8) << p.first << setw(12) << p.second << setw(12) << code[p.first]
             << setw(10) << d << setw(14) << p.second * d << "\n";
        bits += p.second * d;
    }
    long long flat = (long long)log.size() * 8;
    cout << "  B(T) = " << bits << " bits\n";
    cout << "  plain 8-bit log      : " << flat << " bits\n";
```

Listing 9.6 (Continued) *Profitina Laundry: activity selection on the dryer bookings (with a brute-force check), Huffman compression of the day's status log with a verified round trip, and the delivery van as a 0-1 knapsack where greed loses 10% of the revenue. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp and run ./laundry.*

```
    cout << "  Huffman-coded log    : " << bits << " bits\n";
    cout << "  saving               : " << fixed << setprecision(2)
         << 100.0 * (flat - bits) / flat << "%\n";
    // round trip
    string bitstr;
    for (char c : log) bitstr += code[c];
    string back; const Node* cur = root;
    for (char x : bitstr) {
        cur = (x == '1') ? cur->r : cur->l;
        if (cur->leaf) { back += cur->ch; cur = root; }
    }
    cout << "  round trip           : " << (back == log ? "IDENTICAL" : "DIFFERENT") <<
"\n";

    cout << "\n======== PART C: the delivery van, whole parcels only ========\n";
    vector<Parcel> p = {
        {"P1", 14, 210}, {"P2", 21, 273}, {"P3", 30, 330},
        {"P4", 12, 132}, {"P5", 25, 200}
    };
    int cap = 60;
    cout << setw(8) << "parcel" << setw(8) << "kg" << setw(10) << "fee"
         << setw(16) << "fee per kg\n";
    for (auto& x : p)
        cout << setw(8) << x.id << setw(8) << x.kg << setw(10) << x.fee
             << setw(16) << setprecision(3) << (double)x.fee / x.kg << "\n";
    cout << "  van capacity = " << cap << " kg\n\n";
    vector<string> g, o;
    int gv = vanGreedy(p, cap, g);
    int ov = vanOptimal(p, cap, o);
    double fv2 = vanFractional(p, cap);
    cout << "  greedy by fee per kg : ";
    for (size_t i = 0; i < g.size(); ++i) { if (i) cout << " + "; cout << g[i]; }
    cout << "   = " << gv << "\n";
    cout << "  true optimum (DP)    : ";
    for (size_t i = 0; i < o.size(); ++i) { if (i) cout << " + "; cout << o[i]; }
    cout << "   = " << ov << "\n";
    cout << "  greedy leaves " << ov - gv << " on the table ("
         << setprecision(1) << 100.0 * (ov - gv) / ov << "% of the optimum)\n";
    cout << "  if the parcels were bulk detergent that CAN be split, greedy would\n";
    cout << "  be optimal and would earn " << setprecision(2) << fv2 << "\n";
    return 0;
}
```

9.11 Common Mistakes, and How to Avoid Them

1. Writing a greedy algorithm and calling it correct. The commonest mistake by a wide margin. Greed is a hope; the exchange argument is what makes it an algorithm (§9.2.2). If you cannot produce the argument, say so in a comment and keep the brute force.

2. Testing on one instance and concluding. In §9.4.3, two of the three *incorrect* rules got the right answer on the eleven activities of the running example. A single passing instance is not weak evidence — it is no evidence.

3. Sorting by the wrong key. Activity selection sorts by **finish** time. Sorting by start time gives 3 instead of 4 on the running example; sorting by duration fails on a three-activity instance. The key is not interchangeable and there is no intuition that substitutes for the proof.

4. Using a closed interval instead of a half-open one. With $[s_i, f_i]$, an activity ending at 10:00 and one starting at 10:00 "overlap" at a single instant and your algorithm silently rejects a compatible pair. Write $[s_i, f_i)$ and test `s[m] >= f[k]`, never `s[m] > f[k]`.

5. Applying the density rule to a 0-1 knapsack. §9.6.4: it returns 160 where 220 is available, and in 20 000 random instances it was suboptimal 48% of the time at $n = 20$. The tell-tale sign is **unused capacity** in greedy's answer.

6. Believing "close enough" is a bound. The mean ratio of greedy to optimum in §9.6.5 was 0.993; the worst was 0.458. An average is not a guarantee. If you must ship greed, ship the fractional bound alongside it.

7. Building a variable-length code that is not prefix-free. The code $a = 0$, $b = 01$, $c = 1$ compresses beautifully and cannot be decoded (§9.7.3). Always generate codewords from a tree, so the property is structural rather than something you have to remember.

8. Forgetting the one-symbol case in a Huffman implementation. A file of identical bytes yields a single leaf at depth 0, an empty codeword, and a zero-bit output that decodes to nothing. **Listing 9.4** pairs the lone symbol with a dummy; every real compressor must do something equivalent.

9. Forgetting to store the tree. The encoded bits are meaningless without the codebook. A compressed file must carry the frequency table or the tree shape in its header, and that header counts toward the compressed size. The 224 000 bits of §9.7.2 are the payload only.

10. Mis-ordering the priority queue. Huffman needs a **min**-queue. In C++, `std::priority_queue` is a **max**-heap by default, so a comparator that returns `a->freq < b->freq` gives you the two *most* frequent symbols and a pessimal code. **Listing 9.4** writes `a->freq > b->freq` for exactly this reason.

11. Assuming your problem is a matroid. Heredity is usually easy and the **exchange property** is usually where it dies — as it does for activity selection (§9.8.3). Test the exchange property on two small independent sets of different sizes before invoking Theorem 9.15.

12. Using the matroid greedy algorithm to minimize. **Greedy**(M, w) maximizes, and the definition requires **strictly positive** weights. To minimize, transform with $w'(x) = w_{\max} + 1 - w(x)$ as §9.8.8 does, and check that the transformation is legitimate — it is here only because every maximal independent set has the same size (Lemma 9.12).

13. Confusing maximal with maximum. $\{a_3, a_9, a_{11}\}$ cannot be extended and is still one short of the optimum (§9.4.2). "My algorithm cannot add anything more" proves nothing outside a matroid.

14. Expecting greed to survive the addition of weights. Maximizing the *number* of activities is greedy-solvable; maximizing their total *value* is not (Exercise 9.15, where greed returns 10 against an optimum of 100). Check what the objective actually is.

9.12 Chapter Summary

A **greedy algorithm** makes, at each step, the choice that looks best at that moment, judged by a local rule that consults neither the future nor the solutions to subproblems, and never reconsiders (§9.2.1). It is therefore fast — typically $\Theta(n \lg n)$, dominated by a sort — and typically **wrong**: nothing in the form of a greedy algorithm makes it correct, so **the proof is the whole job** (§9.2.2).

Making change shows the point in miniature. The rule "biggest coin that fits" is wrong on {1,3,4} — it pays 6 as $4 + 1 + 1$ where $3 + 3$ suffices, and is wrong at 49 of the first 200 amounts — and wrong on {1,10,25}, where 30 costs six coins instead of three. On {1,5,10,25} and on the Indonesian denominations it is optimal at every amount (**Table 9.1**, **Listing 9.1**). The rule never changed; only the structure did.

Activity selection (**Figure 9.1**) asks for a maximum-size set of mutually compatible activities on one resource. Of four plausible rules, earliest-start and shortest-duration fail on three-activity instances and fewest-conflicts fails on nine (**Listing 9.2**); **earliest finish time** is correct. The proof has two halves. **Optimal substructure** (Theorem 9.1): an optimal A_{ij} splits as $A_{ik} \cup \{a_k\} \cup A_{kj}$ with $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$, and both pieces are optimal, by cut-and-paste (**Figure 9.2**) — which yields the $\Theta(n^3)$ dynamic program Equation (9.4), $c[i,j] = \max_{a_k \in S_{ij}}\{c[i,k] + c[k,j] + 1\}$. **The greedy-choice property** (Theorem 9.2): the earliest finisher of S_k lies in *some* maximum-size solution, proved by the **exchange argument** — take any optimal A_k, swap its first finisher for a_m, and verify the result is feasible and the same size. Together with the observation that only one subproblem remains (**Figure 9.3**) they give Corollary 9.3 and hence **Recursive-Activity-Selector** (**Figure 9.4**), whose trace on the running example (**Figure 9.5**) yields $\{a_1, a_4, a_8, a_{11}\}$, and the iterative **Greedy-Activity-Selector** (**Figure 9.6**). Both are $\Theta(n)$ on sorted input, hence $\Theta(n\lg n)$ overall (**Table 9.3**).

The elements of the greedy strategy are the six steps of **Table 9.4** and two ingredients: the **greedy-choice property** and **optimal substructure**. The practical recipe is: state the objective, list every plausible local rule, try to break each by exhaustive search on small instances, attempt the exchange argument on the survivors, prove optimal substructure by cut-and-paste, and only then write code.

The two knapsacks separate the techniques (**Table 9.7**). Both have optimal substructure. The **fractional** knapsack has the greedy-choice property, and the density rule is optimal (Theorem 9.5) — on **Table 9.5** it earns 240. The **0-1** knapsack does not: the same rule earns 160 where 220 is available (**Table 9.6**), a 27.3% shortfall, and was suboptimal in 48% of random 20-item instances (**Listing 9.3**). The proof of Theorem 9.5 uses divisibility at exactly one step, and that is the whole difference.

Huffman codes (**Figure 9.7**) compress by giving frequent characters short codewords. The example file of 100 000 characters costs 300 000 bits fixed-length and 224 000 variable-length, a 25.33% saving (**Table 9.8**). Codes must be **prefix-free** or they cannot be decoded, and drawing them as **trees** (**Figure 9.8**) makes the property structural. Optimal trees are **full** (Lemma 9.6), and the cost is $B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$ (Equation (9.8)), which also equals the sum of the internal-node labels — that is, the total of all the merges. **Huffman**(C) (**Figure 9.9**) repeatedly merges the two least frequent items; traced on the running example (**Figure 9.10**, **Table 9.9**) the merges are 14,25,30,55,100, summing to 224. Correctness is **Lemma 9.7** (a low frequency and a deep position belong together: swapping them changes the cost by $(a.freq - x.freq)(d(a) - d(x)) \geq 0$, as in Equation (9.9)) and **Lemma 9.8** (collapsing $\{x, y\}$ into z lowers the cost by exactly $x.freq + y.freq$, as in Equation (9.10)), giving Theorem 9.9. The running time is $O(n\lg n)$ with a binary min-heap (Theorem 9.10). **Listing 9.4** encodes and decodes real text 51.5% smaller with an identical round trip.

Matroids (**Figure 9.11**) answer the question "when is greed automatically correct?". A matroid is a finite ground set with a **hereditary** family of *independent* subsets satisfying the **exchange property**: a smaller independent set can always be grown using an element of a larger one. The

consequence that matters is **Lemma 9.12** — in a matroid all maximal independent sets have the same size, so *maximal* and *maximum* coincide, and a greedy algorithm's effortless maximality is enough. **Greedy**(M, w) (**Pseudocode 9.1**) takes elements in decreasing weight order, keeping each that preserves independence, and is optimal for every weighted matroid (Lemmas 9.13, 9.14, Theorem 9.15) in $O(n\lg n + nf(n))$. The **graphic matroid** (**Figure 9.12**) makes it Kruskal's algorithm; **unit-time task scheduling with deadlines and penalties** (**Figure 9.13**) is a matroid by Theorem 9.17, via the test $N_t(A) \leq t$ of Lemma 9.16, and greed returns the early set $\{a_1, a_2, a_3, a_4, a_7\}$ for a minimum penalty of 50 (**Table 9.10**, **Listing 9.5**). Matroid theory covers many practical cases but **not** activity selection or Huffman coding, whose exchange property fails.

The catalog (**Table 9.11**) lists what is coming: Kruskal and Prim in Chapter 11, Dijkstra in Chapter 13, each proved by an exchange argument you have now seen three times.

9.12.1 Looking ahead

Chapter 10 leaves optimization for a while and builds the vocabulary of **graphs**: how to represent them, and how to walk over them with breadth-first and depth-first search. Those two traversals are the substrate on which Chapters 11 and 13 rest.

Chapter 11 then returns to this chapter's material with interest. **Minimum spanning trees** are the canonical matroid problem, and §9.8.8 has already run Kruskal's algorithm on the graph Chapter 11 uses — the answer, weight 37, is there for you to check against. **Prim's algorithm** is a greedy algorithm that is *not* the matroid greedy algorithm, and watching its correctness proof unfold as an exchange argument will be the moment §9.4.7 pays for itself. Chapter 13's **Dijkstra** completes the pattern, and its failure on negative edges is the last and sharpest illustration of §9.6: when the greedy-choice property goes, the table comes back.

9.13 Exercises

9.1 Consider the coin system {1,4,6}. Find, by hand, the smallest amount at which the greedy change rule is not optimal. Give the greedy answer and the optimal answer, with the arithmetic, and explain in one sentence which greedy step did the damage.

9.2 (a) Prove that if 1 is one of the denominations, the greedy change rule always terminates having paid the exact amount. (b) Give a system without 1 and an amount for which the greedy rule gets stuck with money still owed, even though exact change is possible.

9.3 Run the **earliest start time** rule by hand on the eleven activities of the running example (**Figure 9.1**), one step per line, showing at each step which activity is taken and which are discarded. Confirm that it selects three activities, and name the one bad decision.

9.4 Trace the **fewest conflicts** rule by hand on the nine-activity counterexample printed by **Listing 9.2**, namely $a_1 = [4,7)$, $a_2 = [9,13)$, $a_3 = [2,5)$, $a_4 = [1,4)$, $a_5 = [1,2)$, $a_6 = [6,10)$, $a_7 = [0,2)$, $a_8 = [7,15)$, $a_9 = [10,11)$. Count the conflicts of every activity at every stage, and show the rule ends with three. Then find a set of four by the earliest-finish rule.

9.5 Verify that $\{a_2, a_4, a_9, a_{11}\}$ from **Figure 9.1** is a set of mutually compatible activities of size 4, by checking every one of the six pairs. Then explain precisely why **Greedy-Activity-Selector** returns $\{a_1, a_4, a_8, a_{11}\}$ instead, and why this does not contradict Theorem 9.2.

9.6 Prove that the rule "repeatedly take the compatible activity with the **latest start time**" is also optimal for the activity-selection problem. *Hint:* do not repeat the exchange argument; reflect the time axis.

9.7 One processor must run n jobs, job i taking p_i units of time, one after another with no gaps. The **completion time** of a job is the moment it finishes. Prove, by an exchange argument on two adjacent jobs, that scheduling the jobs in non-decreasing order of p_i (**shortest job first**) minimizes the sum of the completion times. Then compute the optimal order and the sum for $p = \langle 7,2,5,1,4,3 \rangle$.

9.8 Construct a family of **two-item** 0-1 knapsack instances on which greedy-by-density obtains an arbitrarily small fraction of the optimum. State the ratio as a function of your parameter.

9.9 Prove that the fractional knapsack problem always has an optimal solution in which **at most one** item is taken fractionally. Then explain what this says about the relationship between the fractional optimum and the 0-1 optimum.

9.10 Build the Huffman code for the eight frequencies $\langle 1,1,2,3,5,8,13,21 \rangle$ — the Fibonacci numbers. List the merges, give the depth of every character, compute $B(T)$, and compare with a fixed-length code. Describe the shape of the tree in one sentence, and say what property of the Fibonacci sequence causes it.

9.11 Prove that in the tree produced by **Huffman**, the two characters of lowest frequency have codewords of maximum length. Is the converse true — must the two characters of maximum codeword length be the two of lowest frequency?

9.12 Suppose the characters are handed to you **already sorted** by frequency. Show how to implement **Huffman** in $O(n)$ time using two ordinary FIFO queues and no heap. State and prove the invariant that makes it work.

9.13 Generalize Huffman coding to codewords over the **ternary** alphabet $\{0,1,2\}$. What does the greedy choice become? For which values of $n = |C|$ does the construction work without modification, and what must be done when it does not? Apply your answer to the frequencies $\langle 5,9,12,13,16,45 \rangle$ of **Figure 9.7**.

9.14 Prove that a full binary tree with ℓ leaves has exactly $\ell - 1$ internal nodes. Give both an induction proof and a counting proof.

9.15 Each activity of **Figure 9.1** now carries a **value**, and we want the compatible set of greatest total value rather than of greatest size. Show by a three-activity example that the earliest-finish rule is no longer optimal, and give a dynamic program that is. What is its running time?

9.16 For a finite set S with $|S| = n$ and an integer $0 \le k \le n$, let $\mathcal{I} = \{A \subseteq S : |A| \le k\}$. Prove that $M = (S, \mathcal{I})$ is a matroid — it is called the **uniform matroid** $U_{k,n}$. What does **Greedy**(M, w) do on it, and what does Theorem 9.15 then say?

9.17 In the task-scheduling instance of **Figure 9.13**, change the penalty of a_5 from 30 to 100, leaving everything else unchanged. Re-run **Greedy** by hand, giving the independence test at every step. State the new early set, the new schedule and the new minimum penalty.

9.18 *(Design task.)* Profitina Laundry buys a second dryer. The bookings are unchanged, but now two may run at once. Design a greedy algorithm for the maximum number of bookings that two identical machines can serve, state precisely the rule for deciding *which* machine a booking goes to, and write the program together with a brute-force check. Does the earliest-finish rule still work?

9.14 Solutions to the Exercises

9.1 Test the amounts in order. For 1,2,3 greedy pays in 1s and so does the optimum. $4 \to 4$ (one coin) ✓. $5 \to 4 + 1$ (two) and no single coin is 5 ✓. $6 \to 6$ (one) ✓. $7 \to 6 + 1$ (two) ✓. Now 8:

- Greedy: largest coin ≤ 8 is 6; owed $8 - 6 = 2$; largest ≤ 2 is 1; owed 1; then 1; owed 0. Coins used: $6 + 1 + 1 = 8$, **three coins**.
- Optimum: $4 + 4 = 8$, **two coins**.

So the smallest failing amount is **8**. **Listing 9.1** confirms it if the line `audit("G", {1, 4, 6}, 200);` is added to `main`: it reports `smallest counterexample: amount = 8`, with greedy `6 + 1 + 1` (3 coins) against the optimal `4 + 4` (2 coins), and greedy wrong for 65 of the 200 amounts. The damaging step was taking the 6: it left a remainder of 2, which in this system can only be paid with two 1s, whereas taking the smaller coin 4 leaves 4, payable with one coin.

9.2 *(a)* Let A be the amount owed. At every step, the largest denomination not exceeding the amount still owed exists, because 1 is a denomination and $1 \leq$ any positive amount owed. Each step reduces the amount owed by at least 1, and the amount owed never becomes negative because we only hand over coins that do not exceed it. So after at most A steps the amount owed is 0, and the payment is exact. ■

(b) Take denominations $\{3,5\}$ and the amount $A = 9$. Greedy takes 5, leaving 4; the largest coin ≤ 4 is 3, leaving 1; now no coin fits and greedy is stuck with 1 still owed. But $9 = 3 + 3 + 3$ exactly. (This is Exercise 7.18's bag-packing failure in another costume.)

9.3 The data is in **Table 9.2**.

- **Step 1.** Earliest start among all eleven: a_3 with $s_3 = 0$. Take $a_3 = [0,6)$. Discard everything overlapping it: $a_1 = [1,4)$ ✗, $a_2 = [3,5)$ ✗, $a_4 = [5,7)$ ✗ (since $5 < 6$), $a_5 = [3,9)$ ✗, $a_6 = [5,9)$ ✗, $a_{10} = [2,14)$ ✗. Survivors: $a_7 = [6,10)$ (as $6 \geq 6$), $a_8 = [8,11)$, $a_9 = [8,12)$, $a_{11} = [12,16)$.
- **Step 2.** Earliest start among the survivors: a_7 with $s_7 = 6$. Take $a_7 = [6,10)$. Discard a_8 $(8 < 10)$ and a_9 $(8 < 10)$. Survivor: a_{11}.
- **Step 3.** Take $a_{11} = [12,16)$. Nothing left.

Result $\{a_3, a_7, a_{11}\}$, **three** activities, matching **Listing 9.2**. The bad decision was step 1: a_3 starts earliest but occupies six time units, blocking six other activities. Starting early is not the same as finishing early.

9.4 First count the conflicts of each activity (two activities conflict when they overlap); **Table 9.12** is the result.

Table 9.12 Conflict counts for the nine activities of Exercise 9.4, before any selection is made. Five activities tie at the minimum of two.

Activity	Interval	Conflicts with	Count
a_1	[4,7)	a_3, a_6	2
a_2	[9,13)	a_6, a_8, a_9	3
a_3	[2,5)	a_1, a_4	2
a_4	[1,4)	a_3, a_5, a_7	3
a_5	[1,2)	a_4, a_7	2
a_6	[6,10)	a_1, a_2, a_8	3
a_7	[0,2)	a_4, a_5	2
a_8	[7,15)	a_2, a_6, a_9	3
a_9	[10,11)	a_2, a_8	2

- **Stage 1.** Minimum count is 2, achieved by a_1, a_3, a_5, a_7, a_9. The implementation takes the first in index order: $a_1 = [4,7)$. Remove a_1 and its conflicts a_3, a_6.
- **Stage 2.** Survivors $a_2, a_4, a_5, a_7, a_8, a_9$. Recount: a_2 conflicts with $a_8, a_9 \to 2$; a_4 with $a_5, a_7 \to 2$; a_5 with $a_4, a_7 \to 2$; a_7 with $a_4, a_5 \to 2$; a_8 with $a_2, a_9 \to 2$; a_9 with $a_2, a_8 \to 2$. All tie at 2; take $a_2 = [9,13)$. Remove a_8, a_9.
- **Stage 3.** Survivors a_4, a_5, a_7, each conflicting with the other two $\to 2$ each. Take $a_4 = [1,4)$. Remove a_5, a_7.
- Nothing remains. Result $\{a_1, a_2, a_4\}$, **three**.

Now the earliest-finish rule. Sorted by finish time: $a_5 = [1,2)$, $a_7 = [0,2)$, $a_4 = [1,4)$, $a_3 = [2,5)$, $a_1 = [4,7)$, $a_6 = [6,10)$, $a_9 = [10,11)$, $a_2 = [9,13)$, $a_8 = [7,15)$, with finish times 2,2,4,5,7,10,11,13,15. Take a_5 ($f_5 = 2$); next with $s \geq 2$ is $a_3 = [2,5)$ ($f_3 = 5$); next with $s \geq 5$ is $a_6 = [6,10)$ ($f_6 = 10$); next with $s \geq 10$ is $a_9 = [10,11)$. Result $\{a_5, a_3, a_6, a_9\}$, **four**. The counterexample construction is of the kind given in [5].

9.5 The six pairs. $a_2 = [3,5)$, $a_4 = [5,7)$, $a_9 = [8,12)$, $a_{11} = [12,16)$.

(a_2, a_4): $s_4 = 5 \geq f_2 = 5$ ✓. (a_2, a_9): $8 \geq 5$ ✓. (a_2, a_{11}): $12 \geq 5$ ✓. (a_4, a_9): $8 \geq 7$ ✓. (a_4, a_{11}): $12 \geq 7$ ✓. (a_9, a_{11}): $12 \geq 12$ ✓. All six compatible, size 4 — and 4 is the maximum, verified by brute force in **Listing 9.2**.

Greedy-Activity-Selector returns $\{a_1, a_4, a_8, a_{11}\}$ because it takes a_1 (the earliest finisher, $f_1 = 4$) rather than a_2 ($f_2 = 5$), and from there the run is forced. There is no contradiction with Theorem 9.2: the theorem says the earliest finisher lies in **some** maximum-size solution, and indeed a_1 lies in $\{a_1, a_4, a_8, a_{11}\}$. It never claimed that every maximum-size solution contains the greedy choice, and $\{a_2, a_4, a_9, a_{11}\}$ is the standing reminder of why that weaker statement is the right one.

9.6 Reflect the time axis. Map each activity $[s_i, f_i)$ to $(-f_i, -s_i]$, or equivalently, choose a constant M larger than every finish time and map $a_i = [s_i, f_i)$ to $\hat{a}_i = [M - f_i,\ M - s_i)$. Three observations complete the proof.

1. The map is a bijection from instances to instances, and it is its own inverse when the same M is used.
2. It preserves compatibility. Two activities are compatible iff their intervals are disjoint; reflection is a rigid motion of the line (a flip followed by a shift), and rigid motions preserve

disjointness. (Half-open intervals map to half-open intervals of the opposite handedness, and the boundary case works out because $s_i \geq f_j \Leftrightarrow M - s_i \leq M - f_j$.)

3. It exchanges starts and finishes: the start of $\hat{a}_i$ is $M - f_i$, so the activity with the **largest** f_i has the **smallest** start after reflection; and the finish of $\hat{a}_i$ is $M - s_i$, so the activity with the **largest** s_i has the **smallest** finish after reflection.

Therefore "take the compatible activity with the latest start time" on the original instance is exactly "take the compatible activity with the earliest finish time" on the reflected instance, which by Theorem 9.4 produces a maximum-size compatible set there; and by observations 1 and 2 the corresponding set in the original instance is compatible and of the same size, hence maximum. ■

9.7 *The exchange.* Let σ be any schedule and suppose two **adjacent** jobs are out of order: job i runs immediately before job j and $p_i > p_j$. Suppose the jobs before them finish at time t. Then

- before the swap: $C_i = t + p_i$ and $C_j = t + p_i + p_j$, contributing $2t + 2p_i + p_j$;
- after the swap: $C_j = t + p_j$ and $C_i = t + p_j + p_i$, contributing $2t + 2p_j + p_i$.

No other job's completion time changes, because the two jobs together still occupy the interval $[t, t + p_i + p_j)$. The change in the total is

$$(2t + 2p_j + p_i) - (2t + 2p_i + p_j) = p_j - p_i < 0,$$

a strict decrease. So any schedule with an adjacent inversion can be strictly improved, and therefore an optimal schedule has no adjacent inversion — which means it is sorted into non-decreasing order of p_i. ■

(A second proof: if job $\sigma(k)$ is the k-th to run, its completion time is $\sum_{r \leq k} p_{\sigma(r)}$, so the total is $\sum_k (n - k + 1)\ p_{\sigma(k)}$: the first job's length is counted n times, the second's $n - 1$ times, and so on. To minimize a sum of products of two sequences, pair the largest multiplier with the smallest length — which is shortest-job-first.)

The computation. With $p = \langle 7,2,5,1,4,3 \rangle$ the sorted order is $j_4(1), j_2(2), j_6(3), j_5(4), j_3(5), j_1(7)$, giving completion times 1,3,6,10,15,22 and total $1 + 3 + 6 + 10 + 15 + 22 = 57$, average $57/6 = 9.5$. **Listing 9.7** checks this against all $6! = 720$ orders:

```
=== E1: one machine, minimize the sum of completion times ===
  job      :   j1   j2   j3   j4   j5   j6
  length   :    7    2    5    1    4    3
  SJF order:   j4   j2   j6   j5   j3   j1
    completion times:    1    3    6   10   15   22
    sum = 57,  average = 9.5000
  all 720 orders examined: best = 57, worst = 97
  SJF attains the best: YES
```

9.8 Let $K \geq 2$ be a parameter. Take capacity $W = K$ and two items:

- item 1: $w_1 = 1$, $v_1 = 2$, density 2;
- item 2: $w_2 = K$, $v_2 = K$, density 1.

Greedy takes item 1 (higher density), leaving capacity $K - 1 < K$, so item 2 no longer fits; greedy's value is 2. The optimum takes item 2 alone, for value K. The ratio is

$$\frac{\text{greedy}}{\text{optimum}} = \frac{2}{K} \to 0.$$

So greedy-by-density has **no** constant approximation guarantee on the 0-1 knapsack. (The standard repair — take the better of the greedy solution and the single most valuable item that fits — does give a guarantee of $1/2$; see [3].)

9.9 Sort the items by density. Suppose an optimal solution takes two items i and j strictly fractionally, $0 < x_i < 1$ and $0 < x_j < 1$, and without loss of generality $v_i/w_i \geq v_j/w_j$. Choose

$$\varepsilon = \min\{(1 - x_i)w_i,\ x_j w_j\} > 0,$$

and shift ε units of weight from item j to item i: increase x_i by ε/w_i and decrease x_j by ε/w_j. The total weight is unchanged, both fractions stay in [0,1], and the value changes by

$$\varepsilon\left(\frac{v_i}{w_i} - \frac{v_j}{w_j}\right) \geq 0.$$

So the new solution is still optimal, and by the choice of ε at least one of x_i, x_j has become 0 or 1 — the number of strictly fractional items has decreased by at least one. Repeating at most n times leaves at most one fractional item. ■

Consequence. The fractional optimum differs from the 0-1 optimum by at most the value of that single partially-taken item, and it is always $\geq$ the 0-1 optimum because every 0-1 solution is a legal fractional solution. So the fractional value computed in $O(n\lg n)$ is an upper bound on the 0-1 optimum, tight to within one item — which is exactly the bound branch-and-bound algorithms use to prune [3].

9.10 The merges, from **Listing 9.7**:

```
=== E2: Huffman with Fibonacci frequencies ===
  frequencies:    1    1    2    3    5    8   13   21
    merge   1 +   1 =   2
    merge   2 +   2 =   4
    merge   3 +   4 =   7
    merge   5 +   7 =  12
    merge   8 +  12 =  20
    merge  13 +  20 =  33
    merge  21 +  33 =  54
  character :    c1    c2    c3    c4    c5    c6    c7    c8
  frequency :     1     1     2     3     5     8    13    21
  depth     :     7     7     6     5     4     3     2     1
  B(T) = 132 bits for 54 characters, i.e. 2.4444 bits per character
  a fixed-length code would need 3 bits each, total 162 bits
```

Check $B(T)$ by hand: $1(7) + 1(7) + 2(6) + 3(5) + 5(4) + 8(3) + 13(2) + 21(1) = 7 + 7 + 12 + 15 + 20 + 24 + 26 + 21 = 132$ ✓. The saving over the fixed-length code is $162 - 132 = 30$ bits, or 18.5%.

The tree is a **caterpillar**: at every level the root has one leaf child and one internal child, so the depths run 1,2,3,4,5,6,7,7 — as unbalanced as a binary tree can be. The cause is the defining property of the Fibonacci numbers, $F_k = F_{k-1} + F_{k-2}$, and the identity it implies, $F_1 + F_2 + \cdots + F_k = F_{k+2} - 1$. After the first k characters have been merged into one node, that node's frequency is $F_{k+2} - 1$. The next two leaves waiting in the queue are F_{k+1} and F_{k+2}, and the node's frequency lies between them: $F_{k+1} \leq F_{k+2} - 1 < F_{k+2}$. So the two smallest items are always the node and the next leaf, and the node is merged again at once. The check on the merges above: the new labels 2,4,7,12,20,33,54 are exactly $F_{k+2} - 1$ for $k = 2, \ldots, 8$. Fibonacci frequencies are the extremal case for Huffman tree depth.

9.11 Huffman merges x and y in its very first iteration, so they become sibling leaves. It remains to show that no leaf is deeper than they are. We prove a stronger statement: *a node extracted from the queue earlier is at least as deep in the final tree as a node extracted later.* (We assume, as **Listing 9.4** does, that ties in frequency are broken in favor of the node created first.)

1. The frequencies extracted from the queue never decrease over the run. At each step the two smallest items are removed, and the only item added is their sum, which is at least as large as either of them; so everything extracted later is at least as large as everything extracted earlier. (This is the invariant proved in Exercise 9.12.)
2. Let u be extracted at merge i and v at a later merge $j > i$, and let p_u and p_v be the internal nodes those merges create — the parents of u and v. Then p_u is created before p_v and, by step 1, $p_u.freq \leq p_v.freq$. With the tie-break above, p_u is therefore extracted before p_v (or p_u is extracted and p_v is the root, which is never extracted).
3. Now argue by induction backwards from the root. Every extracted node's depth is its parent's depth plus one. By step 2 the parents are ordered the same way as the children, so $\text{depth}(p_u) \geq \text{depth}(p_v)$, and adding one to both sides gives $\text{depth}(u) \geq \text{depth}(v)$.

Since x and y are the first two items extracted, no leaf is deeper than they are, and their codewords have maximum length. ■

The converse is false. In **Figure 9.8**(b), the maximum-length codewords are e (frequency 9) and f (frequency 5) — those are indeed the two lowest. But consider frequencies $\langle 1,1,1,1 \rangle$: **all four** characters get depth 2, so the characters of maximum codeword length are all of them, not just two. More sharply, with $\langle 1,1,2,2 \rangle$ **Huffman** produces a balanced tree in which the two frequency-2 characters also have maximum depth. So maximum depth does not single out the two smallest frequencies.

9.12 Use two FIFO queues.

- Q_1 holds the n leaves in non-decreasing frequency order — given to us already sorted.
- Q_2 starts empty and receives each newly created internal node at its **back**.

`Extract-Min` becomes: compare the fronts of Q_1 and Q_2 (treating an empty queue as $+\infty$) and dequeue the smaller. Each of the $2(n-1)$ extractions and $n-1$ insertions is $O(1)$, so the whole construction is $O(n)$.

The invariant that makes it work: Q_2 **is always in non-decreasing order of frequency.** Proof. Let $z_1, z_2, \ldots$ be the internal nodes in creation order, with frequencies $g_1, g_2, \ldots$. We show $g_{k+1} \geq g_k$ for every k.

- At step k the two smallest available items, say of frequencies $u \leq v$, are extracted, so $g_k = u + v$. Every item left behind has frequency at least v, because u and v were the two smallest.
- At step $k+1$ two items are extracted, say of frequencies $u' \leq v'$. Each of them is either an item left behind at step k (frequency $\geq v$) or the new node z_k itself (frequency $u + v \geq v$). So $u' \geq v$ and $v' \geq v$.
- *Case 1: neither is* z_k. Then $g_{k+1} = u' + v' \geq v + v \geq u + v = g_k$.
- *Case 2: one of them is* z_k. Then $g_{k+1} = g_k + (\text{the other}) \geq g_k$, since frequencies are non-negative.

Either way $g_{k+1} \geq g_k$. ■

Since Q_1 is sorted by hypothesis and Q_2 is sorted by the invariant, the global minimum is always at the front of one of them. This is the standard $O(n)$ implementation of Huffman's construction [6].

9.13 *The greedy choice becomes: merge the* ***three*** *least frequent items into one node with three children*, labeled 0, 1 and 2. The cost formula is unchanged, $B(T) = \sum_c c.freq \cdot d_T(c)$, but $d_T(c)$ now counts ternary digits ("trits") rather than bits.

When it works unmodified. A full ternary tree has 3 children per internal node. Each merge removes 3 items and adds 1, a net loss of 2, and we must finish with exactly 1 item. So we need $n - 2m = 1$ for some integer number of merges m, that is, n must be **odd**. In general, for an alphabet of size k the requirement is $n \equiv 1 \pmod{k-1}$.

What to do when it is not. Add dummy characters of frequency 0 until n is odd (in general, until $n \equiv 1 \pmod{k-1}$). Zero-frequency characters contribute nothing to $B(T)$, so the code for the real characters stays optimal; their codewords are simply discarded afterward. Crucially the dummies must be added **before** the first merge, so that they are consumed by the very first (cheapest) merge and do not distort the later ones.

Applied to $\langle 5,9,12,13,16,45 \rangle$. Here $n = 6$, which is even, so add one dummy of frequency 0, giving $\langle 0,5,9,12,13,16,45 \rangle$ with $n = 7$. * Merge the three smallest, $0 + 5 + 9 = 14$. Queue: 12,13,14,16,45. * Merge $12 + 13 + 14 = 39$. Queue: 16,39,45. * Merge $16 + 39 + 45 = 100$. Done. Depths: the characters 45 and 16 are at depth 1; 12 and 13 at depth 2; 5 and 9 at depth 3 (and the dummy, discarded). So

$$\begin{aligned} B(T) &= 45(1) + 16(1) + 12(2) + 13(2) + 5(3) + 9(3) \\ &= 45 + 16 + 24 + 26 + 15 + 27 = 153 \text{ trits.} \end{aligned}$$

Compare with $\lceil \log_3 6 \rceil = 2$ trits fixed-length, costing $2 \times 100 = 200$ trits: a saving of 23.5%. (Had we forgotten the dummy and merged $5 + 9 + 12 = 26$ first, we would have ended with $26,13,16,45 \to 26 + 13 + 16 = 55$, then $55 + 45$ — a node with only **two** children, giving $B = 45(1) + 16(2) + 13(2) + 5(3) + 9(3) + 12(3) = 45 + 32 + 26 + 15 + 27 + 36 = 181$, which is worse by 28 trits. The dummy is not a technicality.)

9.14 *By induction on ℓ.* **Base:** a full binary tree with $\ell = 1$ leaf is a single node, which is a leaf, so it has $0 = \ell - 1$ internal nodes ✓. **Step:** let T be full with $\ell \geq 2$ leaves. Its root is internal (else $\ell = 1$), so by fullness it has two subtrees T_L and T_R, both full, with $\ell_L + \ell_R = \ell$ and $\ell_L, \ell_R \geq 1$. By the inductive hypothesis they have $\ell_L - 1$ and $\ell_R - 1$ internal nodes. Adding the root, T has $(\ell_L - 1) + (\ell_R - 1) + 1 = \ell_L + \ell_R - 1 = \ell - 1$ ✓. ■

By counting edges. Let i be the number of internal nodes and ℓ the number of leaves, so the tree has $n = i + \ell$ nodes and, being a tree, $n - 1$ edges. Every edge is the edge from some node to one of its children, and every internal node contributes exactly 2 such edges while leaves contribute 0. So the number of edges is $2i$. Equating, $2i = i + \ell - 1$, hence $i = \ell - 1$ ✓. ■

9.15 Take three activities: $a_1 = [0,10)$ of value 100, $a_2 = [0,3)$ of value 5, $a_3 = [4,9)$ of value 5. The earliest-finish rule sorts by finish time, taking a_2 ($f = 3$), then a_3 ($s = 4 \geq 3$), for a total value of 10. But a_1 alone is worth 100. From **Listing 9.7**:

```
=== E3: weighted activity selection -- greedy now FAILS ===
  activity :        a1        a2        a3
  [s, f)   :    [0,10)     [0,3)     [4,9)
  value    :       100         5         5
  greedy by earliest finish takes a2+a3, value 10
  dynamic programming finds value 100
  brute force confirms 100
  greedy loses 90 of 100 -- adding weights destroys the greedy-choice property
```

The dynamic program. Sort the activities by finish time. For $1 \le i \le n$ define $q(i)$ as the largest index $j < i$ with $f_j \le s_i$, or 0 if none — the last activity that can precede a_i. Let best$[i]$ be the greatest total value obtainable using only activities $a_1, \dots, a_i$. Then

$$\text{best}[0] = 0, \qquad \text{best}[i] = \max\{\,\text{best}[i-1],\ \ v_i + \text{best}[q(i)]\,\}.$$

The two terms are the two choices: leave a_i out, or take it (in which case everything compatible with it and finishing earlier is contained in $a_1, \dots, a_{q(i)}$). This is exactly the pattern of §7.8. Sorting is $O(n\lg n)$; computing all $q(i)$ by binary search over the sorted finish times is $O(n\lg n)$; the recurrence is $O(n)$. Total: $\Theta(n\lg n)$.

Why greed died. The greedy-choice property failed. The earliest finisher a_2 is not in any optimal solution, so no exchange argument can rescue it: Theorem 9.2's step 5 kept the size the same, but with values the exchange can lose value, and here it loses 95 of 100.

9.16 Three conditions.

1. S is finite. ✓
2. **Hereditary.** If $B \in \mathcal{I}$ so $|B| \le k$, and $A \subseteq B$, then $|A| \le |B| \le k$, so $A \in \mathcal{I}$. ✓ ($\varnothing \in \mathcal{I}$ since $0 \le k$.)
3. **Exchange.** Let $A, B \in \mathcal{I}$ with $|A| < |B|$. Then $|A| < |B| \le k$, so $|A| \le k-1$. Since $|B| > |A|$, the set $B - A$ is non-empty; pick any x in it. Then $|A \cup \{x\}| = |A| + 1 \le k$, so $A \cup \{x\} \in \mathcal{I}$. ✓

So $U_{k,n}$ is a matroid. **Greedy**(M, w) sorts S by decreasing weight and adds elements while the size stays $\le k$, so it returns **the** k **heaviest elements** (all n of them if $k \ge n$). Theorem 9.15 then says — correctly but unexcitingly — that the k heaviest elements form the heaviest subset of size at most k. The uniform matroid is the trivial case, and it is useful precisely because it shows the theorem is not vacuous: sorting-and-taking really is what a matroid greedy algorithm does. Matroids were introduced by Whitney [7]; their exact relationship with greedy algorithms is due to Edmonds [8], and Oxley [9] is the standard modern reference.

9.17 Deadlines are unchanged at $d = \langle 4,2,4,3,1,4,6 \rangle$; penalties are now $w = \langle 70,60,50,40,100,20,10 \rangle$, totalling 350. In decreasing order of penalty the tasks are $a_5(100), a_1(70), a_2(60), a_3(50), a_4(40), a_6(20), a_7(10)$.

- a_5 $(d = 1)$: $\{a_5\}$ has $N_1 = 1 \le 1$ ✓. **Keep.**
- a_1 $(d = 4)$: $\{a_5, a_1\}$ has $N_1 = 1 \le 1$, $N_2 = 1 \le 2$, $N_3 = 1 \le 3$, $N_4 = 2 \le 4$ ✓. **Keep.**
- a_2 $(d = 2)$: $\{a_5, a_1, a_2\}$ has $N_1 = 1 \le 1$, $N_2 = 2 \le 2$ ✓, $N_3 = 2 \le 3$, $N_4 = 3 \le 4$ ✓. **Keep.**
- a_3 $(d = 4)$: $\{a_5, a_1, a_2, a_3\}$ has $N_2 = 2 \le 2$, $N_4 = 4 \le 4$ ✓. **Keep.**
- a_4 $(d = 3)$: adding it gives $N_3 = 3 \le 3$ ✓ but $N_4 = 5 > 4$ ✗. **Reject.**
- a_6 $(d = 4)$: $N_4 = 5 > 4$ ✗. **Reject.**
- a_7 $(d = 6)$: $N_4 = 4 \le 4$, $N_5 = 4 \le 5$, $N_6 = 5 \le 6$ ✓. **Keep.**

Early set $A = \{a_1, a_2, a_3, a_5, a_7\}$ of weight $70 + 60 + 50 + 100 + 10 = 290$; late set $\{a_4, a_6\}$ with penalty $40 + 20 =$ **60**. Check: $290 + 60 = 350$ ✓. Brute force over all 128 subsets confirms 290 is the best achievable early weight, so 60 is the minimum penalty. The schedule is the early tasks in non-decreasing deadline order followed by the late ones: $\langle a_5, a_2, a_1, a_3, a_7, a_4, a_6 \rangle$ — slot 1 a_5 $(d = 1$ ✓$)$, slot 2 a_2 $(d = 2$ ✓$)$, slot 3 a_1 $(d = 4$ ✓$)$, slot 4 a_3 $(d = 4$ ✓$)$, slot 5 a_7 $(d = 6$ ✓$)$, then a_4 and a_6 late.

Note what changed and what did not: raising a_5's penalty from 30 to 100 pulled it to the front of the order and it was kept, but the independence test still allows only four tasks with deadline ≤ 4, so **something** had to be dropped — and it was a_4, now the cheapest of the contenders. The total penalty rose from 50 to 60 even though we only changed a number we were previously paying.

9.18 *The rule.* Sort the bookings by **finish time**, as before. Keep an array freeAt$[1..k]$ of the times at which each machine next becomes free (initially $-\infty$). For each booking in order, consider the machines with freeAt$[m] \leq s$; if there are none, reject the booking; otherwise assign it to the machine whose freeAt is **largest** among those — the "busiest compatible machine". Update that machine's freeAt to f.

Why "busiest compatible", not "any"? Assigning to the machine that has been idle longest wastes the other machine's tighter availability. Taking the largest compatible freeAt leaves the remaining machines as free as possible, which is the same instinct as the earliest-finish rule itself: keep the resource as available as you can.

Does earliest-finish still work? Yes. A set of intervals can be run on k identical machines if and only if no point in time is covered by more than k of them, so the problem is "choose the largest subset of depth at most k" — and the sorted sweep above solves it. Empirically, 300 000 random instances with n between 2 and 9 and $k = 2$ were checked against a brute-force search over all subsets (testing each subset's maximum overlap depth), and the greedy answer matched the optimum on **every one**. The program and the check are a direct extension of Part A of **Listing 9.6**: replace the single `freeAt` scalar by a `vector<int> freeAt(k)` and the test `s >= freeAt` by the busiest-compatible-machine selection above.

For Profitina Laundry. Run on the eleven bookings of §9.10.1, the algorithm serves **7** bookings with two dryers against 4 with one — dryer 1 taking Pak Anto, Warung Bu Tin, Salon Ayu and Klinik Sehat exactly as before, and dryer 2 taking Hotel Melati, Bu Wati and Pak Hendra. Brute force over all 2^{11} subsets (accepting a subset when no instant is covered by more than two bookings) confirms that 7 is the maximum. A third dryer would reach 10 of the 11. That is the argument for buying the second machine, and it is an argument a greedy algorithm can make in $\Theta(n\lg n)$.

Listing 9.7 The programs for Exercises 9.7, 9.10 and 9.15: shortest-job-first checked against all $n!$ orders, Huffman on Fibonacci frequencies, and weighted activity selection where the greedy rule fails. Compile with `g++ -std=c++17 -O2 -o exercises exercises.cpp` and run `./exercises`.

```
// exercises.cpp — the three programs asked for in the exercises.
//   E1  minimizing the average completion time on one machine: shortest job
//       first, checked against every one of the n! orders for small n.
//   E2  Huffman on Fibonacci frequencies: the tree that is as lopsided as a
//       binary tree can be.
//   E3  WEIGHTED activity selection: greedy by earliest finish is now WRONG,
//       and dynamic programming is needed.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <queue>
#include <map>
#include <algorithm>
#include <numeric>
using namespace std;

// ---------------------------------------------------- E1: shortest job first --
long long totalCompletion(const vector<int>& p, const vector<int>& order) {
    long long clock = 0, sum = 0;
    for (int i : order) { clock += p[i]; sum += clock; }
    return sum;
}

void e1() {
    cout << "=== E1: one machine, minimize the sum of completion times ===\n";
    vector<int> p = { 7, 2, 5, 1, 4, 3 };
    int n = (int)p.size();
    cout << "  job      :"; for (int i = 0; i < n; ++i) cout << setw(5) << ("j" +
to_string(i + 1)); cout << "\n";
    cout << "  length   :"; for (int x : p) cout << setw(5) << x; cout << "\n";

    vector<int> sjf(n); iota(sjf.begin(), sjf.end(), 0);
    sort(sjf.begin(), sjf.end(), [&](int a, int b) { return p[a] < p[b]; });
    cout << "  SJF order:"; for (int i : sjf) cout << setw(5) << ("j" + to_string(i + 1));
cout << "\n";
    long long sjfCost = totalCompletion(p, sjf);
    cout << "    completion times:";
    { long long c = 0; for (int i : sjf) { c += p[i]; cout << setw(5) << c; } }
    cout << "\n    sum = " << sjfCost << ",  average = " << fixed << setprecision(4)
         << (double)sjfCost / n << "\n";
```

Listing 9.7 (Continued) *The programs for Exercises 9.7, 9.10 and 9.15: shortest-job-first checked against all n! orders, Huffman on Fibonacci frequencies, and weighted activity selection where the greedy rule fails. Compile with g++ -std=c++17 -O2 -o exercises exercises.cpp and run ./exercises.*

```
    vector<int> order(n); iota(order.begin(), order.end(), 0);
    long long best = -1, worst = -1;
    do {
        long long c = totalCompletion(p, order);
        if (best < 0 || c < best) best = c;
        if (worst < 0 || c > worst) worst = c;
    } while (next_permutation(order.begin(), order.end()));
    cout << "  all " << [&]{ long long f = 1; for (int i = 2; i <= n; ++i) f *= i; return
f; }()
         << " orders examined: best = " << best << ", worst = " << worst << "\n";
    cout << "  SJF attains the best: " << (sjfCost == best ? "YES" : "NO") << "\n\n";
}

// ---------------------------------------------- E2: Huffman on Fibonacci -----
struct Node {
    long long freq; string name; bool leaf; int serial; Node *l, *r;
    Node(long long f, string nm) : freq(f), name(nm), leaf(true),  serial(0), l(nullptr),
r(nullptr) {}
    Node(Node* a, Node* b)       : freq(a->freq + b->freq), name(""), leaf(false),
serial(0), l(a), r(b) {}
};
struct Later {
    bool operator()(const Node* a, const Node* b) const {
        if (a->freq != b->freq) return a->freq > b->freq;
        return a->serial > b->serial;
    }
};
void walk(const Node* n, int d, map<string,int>& depth) {
    if (!n) return;
    if (n->leaf) { depth[n->name] = d; return; }
    walk(n->l, d + 1, depth); walk(n->r, d + 1, depth);
}

void e2() {
    cout << "=== E2: Huffman with Fibonacci frequencies ===\n";
    vector<long long> fib = { 1, 1, 2, 3, 5, 8, 13, 21 };
    vector<string> nm = { "c1", "c2", "c3", "c4", "c5", "c6", "c7", "c8" };
    priority_queue<Node*, vector<Node*>, Later> Q;
    int serial = 0;
    for (size_t i = 0; i < fib.size(); ++i) { Node* x = new Node(fib[i], nm[i]); x->serial
= serial++; Q.push(x); }
    cout << "  frequencies:"; for (long long f : fib) cout << setw(5) << f; cout << "\n";
    while (Q.size() > 1) {
        Node* x = Q.top(); Q.pop();
        Node* y = Q.top(); Q.pop();
        Node* z = new Node(x, y); z->serial = serial++;
        cout << "    merge " << setw(3) << x->freq << " + " << setw(3) << y->freq
             << " = " << setw(3) << z->freq << "\n";
        Q.push(z);
    }
```

Listing 9.7 (Continued) *The programs for Exercises 9.7, 9.10 and 9.15: shortest-job-first checked against all n! orders, Huffman on Fibonacci frequencies, and weighted activity selection where the greedy rule fails. Compile with g++ -std=c++17 -O2 -o exercises exercises.cpp and run ./exercises.*

```
    Node* root = Q.top();
    map<string,int> depth; walk(root, 0, depth);
    long long B = 0;
    cout << "  character :"; for (auto& s : nm) cout << setw(6) << s; cout << "\n";
    cout << "  frequency :"; for (long long f : fib) cout << setw(6) << f; cout << "\n";
    cout << "  depth     :"; for (auto& s : nm) cout << setw(6) << depth[s]; cout << "\n";
    for (size_t i = 0; i < fib.size(); ++i) B += fib[i] * depth[nm[i]];
    cout << "  B(T) = " << B << " bits for " << accumulate(fib.begin(), fib.end(), 0LL)
         << " characters, i.e. " << fixed << setprecision(4)
         << (double)B / accumulate(fib.begin(), fib.end(), 0LL) << " bits per character\n";
    cout << "  a fixed-length code would need 3 bits each, total "
         << 3 * accumulate(fib.begin(), fib.end(), 0LL) << " bits\n\n";
}

// ------------------------------------ E3: WEIGHTED activity selection ------
struct WAct { int s, f, v; };

void e3() {
    cout << "=== E3: weighted activity selection -- greedy now FAILS ===\n";
    vector<WAct> a = { {0, 10, 100}, {0, 3, 5}, {4, 9, 5} };
    int n = (int)a.size();
    cout << "  activity :"; for (int i = 0; i < n; ++i) cout << setw(10) << ("a" +
to_string(i + 1)); cout << "\n";
    cout << "  [s, f)   :"; for (auto& x : a) cout << setw(10) << ("[" + to_string(x.s) +
"," + to_string(x.f) + ")"); cout << "\n";
    cout << "  value    :"; for (auto& x : a) cout << setw(10) << x.v; cout << "\n";

    // greedy by earliest finish, adding up the values it happens to collect
    vector<int> idx(n); iota(idx.begin(), idx.end(), 0);
    sort(idx.begin(), idx.end(), [&](int p, int q) { return a[p].f < a[q].f; });
    int gv = 0, freeAt = -1; string gp;
    for (int i : idx) if (a[i].s >= freeAt) { gv += a[i].v; freeAt = a[i].f; gp +=
(gp.empty() ? "" : "+") + string("a") + to_string(i + 1); }
    cout << "  greedy by earliest finish takes " << gp << ", value " << gv << "\n";

    // dynamic programming: sort by finish, then best[i] = max(best[i-1], v_i + best[j])
    vector<WAct> b = a;
    sort(b.begin(), b.end(), [](const WAct& p, const WAct& q) { return p.f < q.f; });
    vector<int> best(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        int j = 0;                                  // last activity finishing at or
before b[i-1].s
        for (int k = i - 1; k >= 1; --k) if (b[k - 1].f <= b[i - 1].s) { j = k; break; }
        best[i] = max(best[i - 1], b[i - 1].v + best[j]);
    }
    cout << "  dynamic programming finds value " << best[n] << "\n";
```

Listing 9.7 (Continued) *The programs for Exercises 9.7, 9.10 and 9.15: shortest-job-first checked against all n! orders, Huffman on Fibonacci frequencies, and weighted activity selection where the greedy rule fails. Compile with g++ -std=c++17 -O2 -o exercises exercises.cpp and run ./exercises.*

```
    int brute = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        vector<int> c; int v = 0;
        for (int i = 0; i < n; ++i) if (mask >> i & 1) { c.push_back(i); v += a[i].v; }
        bool ok = true;
        for (size_t i = 0; i + 1 < c.size() && ok; ++i)
            for (size_t j = i + 1; j < c.size() && ok; ++j)
                if (!(a[c[i]].s >= a[c[j]].f || a[c[j]].s >= a[c[i]].f)) ok = false;
        if (ok) brute = max(brute, v);
    }
    cout << "  brute force confirms " << brute << "\n";
    cout << "  greedy loses " << brute - gv << " of " << brute
         << " -- adding weights destroys the greedy-choice property\n";
}

int main() { e1(); e2(); e3(); return 0; }
```

For further practice on greedy problems of exactly this kind — interval scheduling, coin systems, Huffman variants — the problem archives cataloged in [10] are the place to go, and the treatment in [4] carries additional exercises on matroids and on the relationship between greedy algorithms and linear programming.

References

[1] Bank Indonesia, “Gambar Uang Rupiah” (catalog of the rupiah banknotes and coins in circulation: 2022 banknote series and 2016 coin series). [Online]. Available: https://www.bi.go.id/id/rupiah/gambar-uang/default.aspx (accessed Sep. 24, 2026).

[2] D. Pearson, “A polynomial-time algorithm for the change-making problem,” *Operations Research Letters*, vol. 33, no. 3, pp. 231–234, May 2005.

[3] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. Chichester, U.K.: Wiley, 1990.

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

[5] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[6] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.

[7] H. Whitney, “On the abstract properties of linear dependence,” *American Journal of Mathematics*, vol. 57, no. 3, pp. 509–533, Jul. 1935.

[8] J. Edmonds, “Matroids and the greedy algorithm,” *Mathematical Programming*, vol. 1, no. 1, pp. 127–136, Dec. 1971.

[9] J. Oxley, *Matroid Theory*, 2nd ed. Oxford, U.K.: Oxford Univ. Press, 2011.

[10] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, 4th ed. Raleigh, NC, USA: Lulu Press, 2020.

Chapter 10

Elementary Graph Algorithms: Breadth-First and Depth-First Search

By the end of this chapter you will be able to look at almost any tangle of things-and-connections — a road map, a circuit board, a family tree, a list of tasks with an order they must be done in, the pages of a website and the links between them — and write it down as a graph, using the right word for every part of it. You will be able to store that graph in a computer in three different ways, say exactly how much memory each way costs, say exactly how long each of the four basic questions takes to answer in each way, and choose between them with a reason rather than a habit. You will be able to run breadth-first search and depth-first search by hand, one step at a time, writing down the queue or the stack after every single step, and get the same answer the machine gets. You will be able to *prove* that breadth-first search finds shortest paths in a graph whose edges have no lengths, rather than merely believing it. You will be able to derive $O(V + E)$ for both searches from first principles. And you will be able to use the two searches to answer six separate questions that at first sight have nothing to do with searching: how many pieces is this graph in, does it contain a cycle, can its vertices be split into two camps with no edge inside a camp, what is the shortest route from here to there, is this set of precedences self-contradictory, and in what order may these tasks be performed.

10.1 Where We Left Off

The nine chapters behind us have all had one thing in common: the input was a **sequence**. In Chapter 2 we sorted an array $A[1..n]$ and searched it. In Chapter 3 we learned to count the cost of doing so, and to solve the recurrences that counting produces. In Chapters 5, 6 and 7 the input grew more structure — a heap is an array read as a tree, a binary search tree is a set of keys arranged so that a comparison halves the search, a dynamic-programming table is a grid of subproblems — but in every case the *structure was ours to choose*. We imposed it. The array did not come with a shape; we gave it one.

Chapter 9 changed the style of argument rather than the shape of the data. A greedy algorithm makes the locally best choice and never reconsiders, and the work of the chapter was proving that for certain problems — activity selection, Huffman coding — the locally best choice is also globally safe.

This chapter changes the shape of the data, and it is the biggest such change in the book. From here to the end, the input **arrives** with a structure already in it, one we did not choose and cannot rearrange. A road network is a fact about Surabaya. The links between web pages are a fact about the web. The precedences among the tasks of a build system are a fact about the software. Our job is no longer to impose an order but to *discover* what order is already there.

The mathematical object for data of that kind is the **graph**, and the two algorithms that everything else in graph theory is built out of are **breadth-first search** and **depth-first search**. Chapters 11 through 15 — minimum spanning trees, topological sort in anger, single-source shortest paths, all-pairs shortest paths — are all refinements, decorations, or weighted

generalizations of the two searches in this chapter. If you understand this chapter completely, the rest of the book's graph material is a series of small steps. If you do not, none of it will make sense. It is worth going slowly.

10.1.1 What is new, and what carries over

Three things carry over unchanged, and it is worth naming them so you know you are not starting from nothing.

The **asymptotic notation** of Chapter 3 is the same notation. When we say a search costs $O(V + E)$ we mean exactly what we meant when we said insertion sort costs $O(n^2)$: there exist constants c and n_0 such that for all inputs of size at least n_0 the running time is at most c times the stated function. The only novelty is that the "size" of the input now has *two* numbers in it rather than one, because a graph can be big in two independent ways — many vertices, or many edges.

The **queue and the stack** of Chapter 2 are the same queue and the same stack. Breadth-first search is a queue with a graph attached; depth-first search is a stack with a graph attached. That sentence is very nearly the whole chapter.

The **loop invariant** discipline of Chapter 2 is the same discipline. When we prove that breadth-first search computes shortest distances, the proof is an induction over the steps of the `while` loop, exactly as the insertion-sort proof was an induction over the steps of its `for` loop.

What is genuinely new is this: a graph algorithm can visit the same vertex twice, and if it does, it may never stop. Every algorithm in this chapter therefore carries a *marker* — a color — whose only job is to remember what has already been looked at. Forgetting the marker is the single most common way to write a graph program that hangs forever, and §10.11 will name it as the first of the common mistakes.

10.2 What a Graph Is

10.2.1 Start with something you can see

Before any symbol, a picture you already know.

Draw five houses in a *kampung*. Between some pairs of houses there is a footpath; between others there is not, because a wall or a drain is in the way. Now stop thinking about the houses and the footpaths as houses and footpaths. Keep only two facts:

1. there are five *things*;
2. certain *pairs* of those things are joined.

Everything else — how big the houses are, how long the footpaths are, who lives where — has been thrown away. What is left is a graph. The houses are called **vertices** (one of them is a *vertex*; some books say *nodes*), and the footpaths are called **edges** (some books say *arcs* or *links*).

That is the whole idea, and it is worth pausing on how violent the simplification is. A graph does not know where its vertices are. When you draw the same graph twice, once with the vertices in a row and once with them in a circle, you have drawn *the same graph*. The picture is a convenience for human eyes; the graph is only the list of things and the list of joined pairs.

KEY IDEA. A drawing of a graph is not the graph. Two drawings that look completely different can be the same graph, and the algorithms in this chapter cannot tell them apart, because they only ever ask "is this pair joined?"

10.2.2 Writing it down

Now we need symbols, and we will introduce each one in words before we use it.

Call the set of things V. The letter V is for *vertices*. A set is just a collection with no repeats and no order, so writing $V = \{a, b, c, d, e\}$ says "there are five things, and their names are a through e" — nothing more.

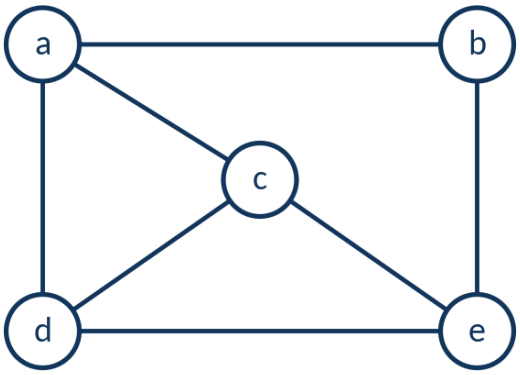
Call the set of joined pairs E. The letter E is for *edges*. Each member of E is a pair of members of V. If the things a and b are joined, then the pair (a, b) is a member of E, which we write $(a, b) \in E$; the symbol $\in$ is read "is a member of" or, more briefly, "is in".

The number of vertices is written $|V|$, read "the size of V" or "the number of vertices". Likewise $|E|$ is the number of edges. Inside asymptotic notation almost everybody, including this book, drops the bars and writes $O(V + E)$ rather than $O(|V| + |E|)$; it is an abuse of notation, universally understood, and it means $O(|V| + |E|)$.

10.2.3 The formal definition

With those three symbols in hand the definition is one line, and **Figure 10.1** states it together with the example we will use for the rest of the chapter.

DEFINITION. A **graph** $G = (V, E)$ consists of a set V of **vertices** and a set $E \subseteq V \times V$ of **edges**, where each edge $e = (u, v)$ is a pair of vertices. If the pair is *ordered* — that is, if (u, v) and (v, u) mean different things — then G is a **directed graph**. If the pair is unordered, so that (u, v) and (v, u) are the same edge, then G is an **undirected graph**.



$$V = \{a, b, c, d, e\}$$

$$E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$$

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.

Two pieces of the definition need unpacking.

$V \times V$, read "V cross V", is the set of *all* pairs you could form from V — for our five vertices that is $5 \times 5 = 25$ pairs, counting (a, a) and counting (a, b) and (b, a) separately. Writing $E \subseteq V \times V$ ("E is a subset of V cross V") therefore says only this: the edges are *some* of the possible pairs. Which ones, the definition does not say — that is the data.

The word **ordered** is the whole difference between the two kinds of graph, and it is worth a concrete example rather than an abstract one. On a two-way street, "there is a road from the

market to the mosque" and "there is a road from the mosque to the market" are the same fact: one edge, unordered. On a one-way street they are different facts: you may need two edges, or one, and which direction it points matters enormously. A graph of two-way streets is undirected; a graph of one-way streets is directed. When we draw a directed graph we put an arrowhead on each edge; when we draw an undirected graph we do not.

Let us read **Figure 10.1** out loud, slowly, because everything later depends on being able to do this without effort.

- "V equals the set containing a, b, c, d and e." Five vertices.
- "E equals the set containing the pair a–b, the pair a–c, the pair a–d, the pair b–e, the pair c–d, the pair c–e, and the pair d–e." Seven edges.
- Because nothing says otherwise, the pairs are unordered: this is an undirected graph. The pair (a, b) and the pair (b, a) are the same edge, and the figure draws it once.
- Which pairs are *missing*? (a, e) and (b, c) and (b, d). Of the $\binom{5}{2} = 10$ unordered pairs available, seven are present and three are absent.

That last line is the one students most often skip, and it is the one that matters: a graph is as much a statement about which pairs are *not* joined as about which are.

10.2.4 Two small conventions

A **self-loop** is an edge from a vertex to itself, (v, v). Self-loops are forbidden in most undirected settings and allowed in directed ones; the depth-first search example of §10.8 contains exactly one, the arc $z \to z$, and we shall see that the algorithm handles it without any special case.

A graph with no self-loops and no repeated edges is called **simple**. Every undirected graph in this chapter is simple unless it says otherwise. When a graph does allow two different edges between the same pair of vertices it is a **multigraph**; a multigraph is what you get if you model two parallel bridges over the same river as two edges rather than one.

10.3 What Graphs Are For

We begin with applications, because that is the right order: nobody should learn a notation before learning what the notation is *for*. **Figure 10.2** and **Figure 10.3** show two of them. We will take the list one item at a time and say, for each, what the vertices are, what the edges are, and which question you would ask.

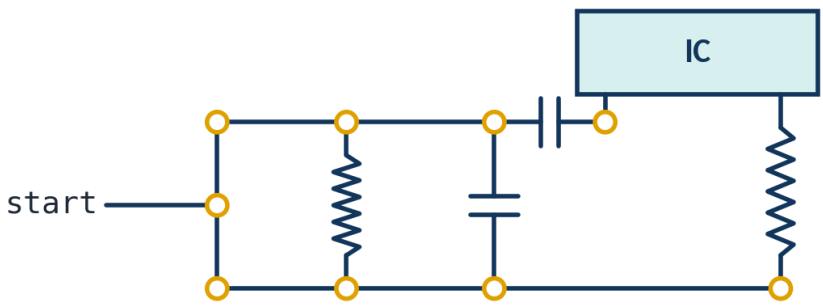


Figure 10.2 An electronic circuit as a graph. The vertices are the solder points where components meet; the edges are the components — resistors, capacitors — that join them. The question "what is the path of least resistance from `start` to the block marked IC?" is a shortest-path question on this graph.

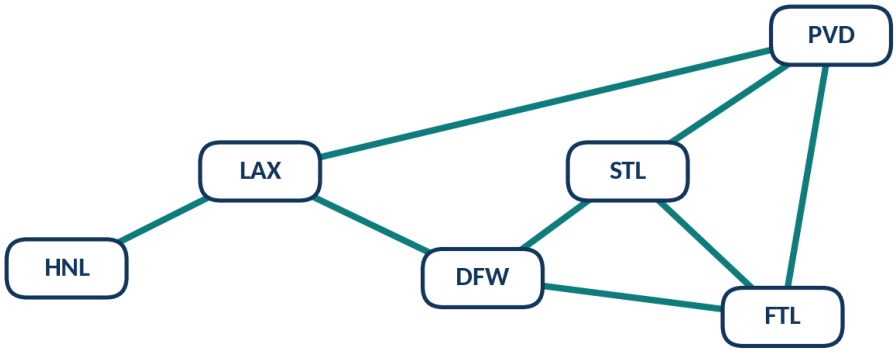


Figure 10.3 A flight network as a graph: the vertices are airports (HNL, LAX, DFW, STL, FTL, PVD), the edges are the routes flown between them. "Can I get from Honolulu to Providence, and with how few changes?" is precisely the question breadth-first search answers in §10.7.

Electronic circuits. Vertices are the junctions where wires and components meet; edges are the components. Because a resistor has a resistance, the edges here carry a number, and the graph is *weighted* (§10.4.14). The question posed in **Figure 10.2** — find the path of least resistance — is a shortest-path problem, and Chapter 13 is devoted to it. But two of this chapter's plainer questions are also asked of circuits every day: *is the circuit connected*, meaning does current have any route at all from the supply to each component, and *does it contain a cycle*, which for a circuit is the difference between a series chain and a loop.

Networks: roads, flights, communication. Vertices are places, edges are the links between them. This is the archetype and the one to hold in your head. The vertices of a road network are junctions; the edges are the stretches of road between junctions. For a flight network, as in **Figure 10.3**, the vertices are airports and the edges are routes. For a communications network the vertices are routers and the edges are the cables or radio links between them. The questions are: *can this reach that*, *by the fewest hops*, *by the shortest distance*, *what breaks if this link fails*.

Notice that these two are not the same question, and the difference organizes the rest of the book. "Fewest hops" counts edges and ignores their lengths — that is this chapter, and breadth-first search answers it. "Shortest distance" adds up lengths — that is Chapter 13, and it needs Dijkstra's algorithm.

Compilers. When a compiler must decide the order in which to evaluate the parts of an expression, or a build system must decide the order in which to compile the files of a program, the vertices are the tasks and each edge $u \to v$ means "u must be finished before v may begin". The graph is directed. The question is: *give me an order of all the tasks that respects every edge*, and its answer is the **topological sort** of §10.9.6. A second question is just as important: *is the set of precedences even satisfiable?* If task u must precede v, and v must precede w, and w must precede u, then no order exists, and the compiler must say so rather than loop. Detecting that is cycle detection, §10.9.2.

Graphics. A polygon mesh is a graph: the vertices are the corners of the triangles and the edges are the triangle sides. Rendering, smoothing, simplifying and texturing a mesh are all graph traversals. A flood fill — the paint-bucket tool in any drawing program — is literally breadth-first search over a graph whose vertices are pixels and whose edges join pixels that touch.

Maze-solving. The vertices are the junctions and dead ends of the maze; the edges are the corridors. Depth-first search is the algorithm you use without being told: walk forward until you cannot, then back up to the last junction with an unexplored corridor and try that. §10.8 makes this precise, and its classic image — a string and a can of paint — is one of the best in the subject.

Mapping. A map application stores the road network as a graph and answers routing queries on it. But it also answers questions this chapter can handle on its own: which addresses are reachable at all, which region of the map is one connected piece, how far in *turns* rather than in kilometers is one place from another.

Routing, searching, clustering in networks. Routing: find a path, or the best path. Searching: starting from one machine, visit every machine reachable from it — a graph search, exactly. Clustering: break the graph into groups of vertices that are densely joined inside and sparsely joined between. The crudest and most useful clustering is *the connected components*, and §10.9.1 computes them with nothing but breadth-first search.

> **EXAMPLE.** Profitina's own AI-visibility scanner meets a graph on every job it runs. When it scans a website it must visit every page reachable from the home page by following links. The vertices are pages; the edges are hyperlinks, and they are *directed*, because a link from page p to page q does not give you a link back. "Visit every page reachable from the home page, and do not visit any page twice" is a graph search and nothing else; §10.10 says what Profitina does about it.

The pattern in all nine of these is worth stating on its own.

> **KEY IDEA.** To model a problem as a graph, answer three questions in this order. **What are the things?** They become V. **When are two things joined?** That rule becomes E. **Does the joining have a direction?** If yes, the graph is directed. Only after those three are answered should you ask which algorithm to run — and if the three answers are wrong, no algorithm will save you.

10.4 The Vocabulary of Graphs

Graph theory is unusual among the subjects in this book in that almost all of its difficulty for a beginner is *vocabulary*: once you know what the words mean, the theorems are short. So we shall do this properly — every term defined, every term illustrated on the chapter's own graph, and a worked example immediately after each definition.

Throughout this section, G is the graph of **Figure 10.1**: $V = \{a, b, c, d, e\}$, $E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$.

10.4.1 Adjacent, incident, neighbor

> **DEFINITION.** Two vertices are **adjacent** if there is an edge joining them. An edge is **incident** on each of the two vertices at its ends. The vertices adjacent to v are the **neighbors** of v, and the set of them is written $\mathrm{Adj}[v]$.

So *adjacent* is a relation between two vertices, and *incident* is a relation between an edge and a vertex. Students mix them up for about a week and then never again.

On G: a and b are adjacent, because $(a, b) \in E$. The vertices a and e are **not** adjacent, because no pair (a, e) appears in E — look for it in **Figure 10.1** and you will not find it. The edge (a, b) is incident on a and incident on b, and on nothing else. And

$$\mathrm{Adj}[a] = \{b, c, d\}, \quad \mathrm{Adj}[b] = \{a, e\}, \quad \mathrm{Adj}[c] = \{a, d, e\}, \quad \mathrm{Adj}[d] = \{a, c, e\}, \quad \mathrm{Adj}[e] = \{b, c, d\}.$$

Read the first of those aloud: "the neighbors of a are b, c and d." Check it against the picture. Then check the other four. This is not busy-work: $\text{Adj}[v]$ is the *only* thing every algorithm in this chapter ever asks the graph for, so the five lines above are, in a real sense, the entire input to the rest of the chapter.

In a directed graph, "adjacent" acquires a direction. We say v is adjacent *to* u when the arc (u, v) exists, and then $\text{Adj}[u]$ holds the heads of the arcs leaving u. Note that in a directed graph $v \in \text{Adj}[u]$ does **not** imply $u \in \text{Adj}[v]$; in an undirected graph it always does, and that asymmetry is the source of half the bugs in graph code.

10.4.2 Degree

DEFINITION. The **degree** of a vertex v in an undirected graph, written $\deg(v)$, is the number of edges incident on v — equivalently, in a simple graph, the number of neighbors it has: $\deg(v) = |\text{Adj}[v]|$.

Figure 10.4 pictures degree: each vertex of G is labeled with its own degree rather than its name.

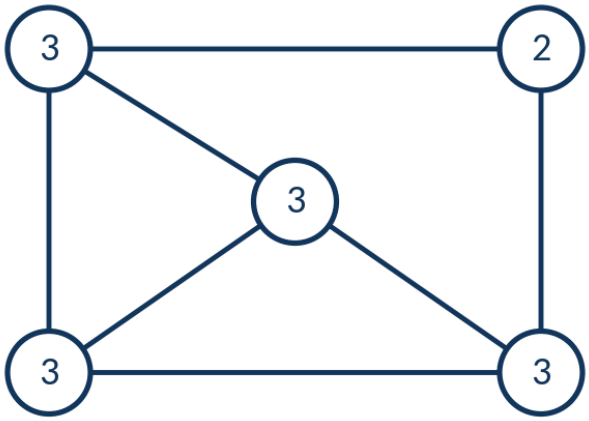


Figure 10.4 Degree. Each vertex of the chapter's graph is labeled with its degree instead of its name. Reading anticlockwise from the top left the labels are 3,2,3,3,3; they sum to 14, which is twice the number of edges.

Let us compute all five degrees by hand, counting the edge-ends at each vertex.

- $\deg(a)$: the edges (a, b), (a, c), (a, d) touch a. That is 3.
- $\deg(b)$: the edges (a, b), (b, e) touch b. That is 2.
- $\deg(c)$: the edges (a, c), (c, d), (c, e) touch c. That is 3.
- $\deg(d)$: the edges (a, d), (c, d), (d, e) touch d. That is 3.
- $\deg(e)$: the edges (b, e), (c, e), (d, e) touch e. That is 3.

Adding them: $3 + 2 + 3 + 3 + 3 = 14$. And $|E| = 7$, so the sum of the degrees is $2|E|$. That is not a coincidence, and the reason can be stated exactly. Let us prove it.

10.4.3 The handshake lemma, proved

HANDSHAKE LEMMA. In any undirected graph $G = (V, E)$,

$$\sum_{v \in V} \deg(v) = 2\,|E|.$$

(10.1)

Proof. We count the same collection of objects in two different ways and equate the two counts. The objects are **edge-ends**: an edge-end is a pair consisting of an edge together with one of the two vertices at its ends. The edge (a, b) therefore contributes two edge-ends, namely $\langle (a, b), a \rangle$ and $\langle (a, b), b \rangle$.

First count, by edges. Every edge has exactly two ends, no more and no fewer, because an edge is by definition a pair of vertices. There are $|E|$ edges. So the number of edge-ends is exactly $2|E|$.

Second count, by vertices. Fix a vertex v. The edge-ends whose vertex is v are in one-to-one correspondence with the edges incident on v, and there are $\deg(v)$ of those, by the definition of degree. Every edge-end has exactly one vertex, so no edge-end is counted at two different vertices and none is missed. Summing over all vertices, the number of edge-ends is $\sum_{v \in V} \deg(v)$.

The two counts count the same set, so they are equal:

$$\sum_{v \in V} \deg(v) = 2\,|E|. \qquad \blacksquare$$

In plain language, Equation (10.1) says: *since adjacent vertices each count the adjoining edge, every edge gets counted twice.* Walk along the edge (c, e): it is counted once when you tally c's neighbors and once again when you tally e's. Seven edges, each counted twice, fourteen.

The lemma has three consequences we shall use.

Consequence 1 — the number of odd-degree vertices is even. Split the sum into the even-degree vertices and the odd-degree ones. The even part is even. The whole sum is $2|E|$, which is even. So the odd part must be even too; and a sum of odd numbers is even only when there is an even *number* of them. (This is why, at a party, the number of people who have shaken hands an odd number of times is always even — hence the lemma's name.)

Consequence 2 — the total length of all the adjacency lists is $2|E|$. This is the same statement wearing a programmer's hat, and it is the single most important fact for the running-time analyses later in the chapter. If we store the graph as $|V|$ lists, one per vertex, with $\mathrm{Adj}[v]$ holding v's neighbors, then list v has $\deg(v)$ entries, so all the lists together hold $\sum_v \deg(v) = 2|E|$ entries. An algorithm that walks every adjacency list exactly once therefore does $\Theta(E)$ work, not $\Theta(V \cdot E)$. Both breadth-first and depth-first search do exactly that, which is where their $O(V + E)$ comes from.

Consequence 3 — the average degree is $2|E|/|V|$. Divide both sides by $|V|$. For our graph that is $14/5 = 2.8$.

10.4.4 In-degree and out-degree

In a directed graph, an arc leaving v and an arc entering v are different things, so degree splits into two.

> **DEFINITION.** In a directed graph, the **out-degree** of v, written $\deg^+(v)$, is the number of arcs whose *tail* is v — that is, the number of arcs of the form $(v, \cdot)$. The **in-degree**, written $\deg^-(v)$, is the number of arcs whose *head* is v — the arcs of the form $(\cdot, v)$. The **degree** of v is $\deg^+(v) + \deg^-(v)$.

The directed analog of the handshake lemma is simpler, and comes with two halves:

$$\sum_{v \in V} \deg^{+}(v) = |E|, \qquad \sum_{v \in V} \deg^{-}(v) = |E|, \qquad \text{so} \qquad \sum_{v \in V} (\deg^{+}(v) + \deg^{-}(v)) = 2|E|. \tag{10.2}$$

Proof. Every arc has exactly one tail, so summing the out-degrees counts every arc exactly once. Every arc has exactly one head, so summing the in-degrees also counts every arc exactly once. ■

Worked example, on the directed graph of §10.8's depth-first search example, whose arcs are $u \to v, u \to x, v \to y, w \to y, w \to z, x \to v, y \to x$ and $z \to z$.

Table 10.1 In- and out-degrees of the directed graph used in §10.8. Both columns sum to $8 = |E|$, as the lemma requires. The self-loop $z \to z$ contributes one to the out-degree of z and one to its in-degree.

Vertex	deg⁺	Which arcs leave	deg⁻	Which arcs enter
u	2	$u \to v, u \to x$	0	—
v	1	$v \to y$	2	$u \to v, x \to v$
w	2	$w \to y, w \to z$	0	—
x	1	$x \to v$	2	$u \to x, y \to x$
y	1	$y \to x$	2	$v \to y, w \to y$
z	1	$z \to z$	1	$z \to z$

Out-degrees: $2 + 1 + 2 + 1 + 1 + 1 = 8$. In-degrees: $0 + 2 + 0 + 2 + 2 + 1 = 8$. And $|E| = 8$. The arithmetic checks, exactly as Equation (10.2) predicts.

A vertex with in-degree 0 is called a **source**; a vertex with out-degree 0 is a **sink**. In **Table 10.1**, u and w are sources and there is no sink. Sources matter: §10.9.6 shows that a topological sort of a directed acyclic graph can be produced by repeatedly removing a source, and a directed acyclic graph always has at least one.

10.4.5 Walks, paths, and simple paths

Here the terminology deserves a note, because books differ and the difference trips people up.

> **DEFINITION.** A **path** (some books say *walk*) from v_1 to v_k is a sequence of vertices $v_1, v_2, \ldots, v_k$ in which every consecutive pair v_i and v_{i+1} is adjacent. Its **length** is the number of edges it uses, which is $k - 1$ — not the number of vertices. A path is **simple** if no vertex is repeated.

In short: a *path* is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent; a *simple path* has no repeated vertices. **Figure 10.5** shows two examples and **Figure 10.6** a third.

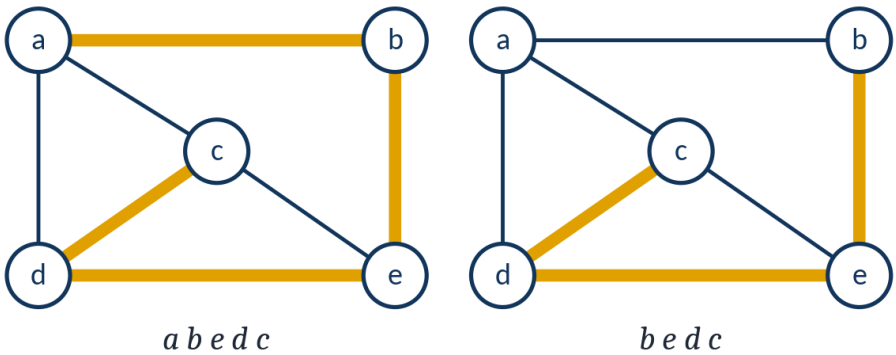


Figure 10.5 Two paths in the chapter's graph, highlighted. On the left, $a\ b\ e\ d\ c$, of length 4; on the right, $b\ e\ d\ c$, of length 3. Both are simple: no vertex is used twice.

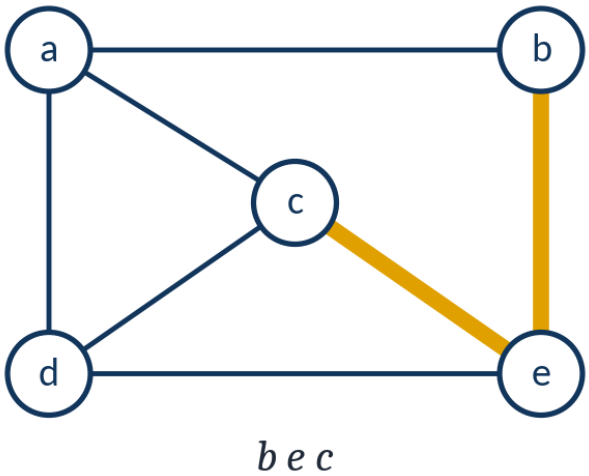


Figure 10.6 A third path, $b\ e\ c$, of length 2. It is the shortest path from b to c in this graph — there is no edge (b, c), so no path of length 1 exists.

Let us verify the left-hand path of **Figure 10.5** edge by edge, because this is exactly the check every algorithm in the chapter performs.

- Is a adjacent to b? Yes: $(a, b) \in E$. Step taken, length so far 1.
- Is b adjacent to e? Yes: $(b, e) \in E$. Length so far 2.
- Is e adjacent to d? Yes: $(d, e) \in E$; the pair is unordered, so (e, d) is the same edge. Length so far 3.
- Is d adjacent to c? Yes: $(c, d) \in E$. Length so far 4.
- Are any vertices repeated in a, b, e, d, c? No. So the path is simple.

Now a path that is *not* simple: a, b, e, c, a, d. Every consecutive pair is adjacent — check them: (a, b), (b, e), (c, e), (a, c), (a, d), all in E — so it is a path of length 5. But a appears twice, so it is not simple. And notice that it is also *pointless*: the shorter path a, d has the same two endpoints and length 1. That observation generalizes into a lemma we shall use in the shortest-path proof:

> **KEY IDEA.** If there is a path from u to v at all, then there is a *simple* path from u to v, and it is no longer. Reason: if some vertex w occurs twice in the path, delete everything strictly between the two occurrences, together with one of them. The result is still a path — the vertex before the first occurrence is still adjacent to w, and w is still adjacent to the vertex after the second — and it is strictly shorter. Repeat until no vertex occurs twice; the process must stop, because the length decreases every time and cannot go below zero.

The consequence we shall lean on is that a simple path in a graph with $|V|$ vertices has length at most $|V| - 1$, because it cannot use a vertex twice. So no shortest path is ever longer than $|V| - 1$ edges — a fact worth remembering when you size an array.

Two more words. A vertex v is **reachable** from u if there is a path from u to v; and we write $\delta(u, v)$, "delta of u and v", for the **shortest-path distance** from u to v, meaning the smallest length of any path from u to v, or ∞ if v is not reachable from u. Every vertex is reachable from itself by the path of length 0 consisting of that vertex alone, so $\delta(v, v) = 0$ always.

10.4.6 Cycles and acyclic graphs

> **DEFINITION.** A **cycle** is a path $v_1, v_2, \ldots, v_k, v_1$ that is simple except that the last vertex equals the first. A graph with no cycles is **acyclic**.

In short: a *cycle* is a simple path, except that the last vertex is the same as the first vertex. **Figure 10.7** pictures one; the small arrows trace the walk, which ends exactly where it began.

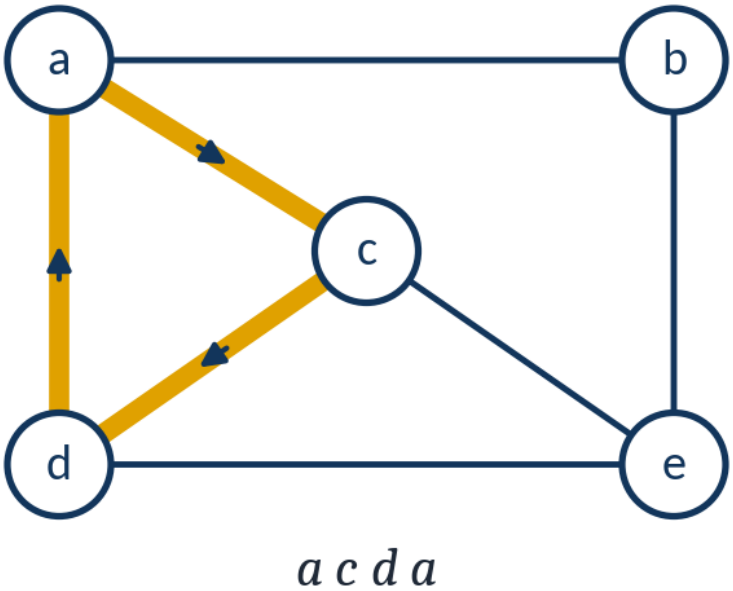


Figure 10.7 A cycle in the chapter's graph, highlighted: $a\ c\ d\ a$. Start at a, walk to c, walk to d, walk back to a — three edges, and you are where you began without having repeated any vertex on the way.

Verify it: $(a, c) \in E$, $(c, d) \in E$, $(a, d) \in E$. Three edges, three vertices, and the only repetition is the first vertex reappearing at the end. That is a cycle of length 3, also called a **triangle**.

Our graph has several other cycles: $a\ b\ e\ c\ a$ (length 4), $a\ b\ e\ d\ a$ (length 4), $c\ d\ e\ c$ (length 3), and the length-5 cycle $a\ b\ e\ c\ d\ a$. So this graph is very much *not* acyclic.

In an undirected simple graph the shortest possible cycle has length 3: length 1 would be a self-loop, which is forbidden, and length 2 would need two distinct edges between the same pair, which a simple graph forbids too. In a **directed** graph, by contrast, both are possible: a self-loop $z \to z$ is a directed cycle of length 1, and the pair of arcs $u \to v$, $v \to u$ is a directed cycle of length 2. This matters in §10.9.2, where the test for a cycle turns out to be genuinely different in the two settings.

A directed graph with no directed cycle is so important that it has its own name and its own acronym: a **directed acyclic graph**, or **DAG**. §10.9.5 is about them.

10.4.7 Connected graphs

DEFINITION. An undirected graph is **connected** if for every pair of vertices u and v there is a path from u to v. A graph that is not connected is **disconnected**.

In short: a graph is *connected* when any two vertices are connected by some path. **Figure 10.8** compares the two situations side by side.

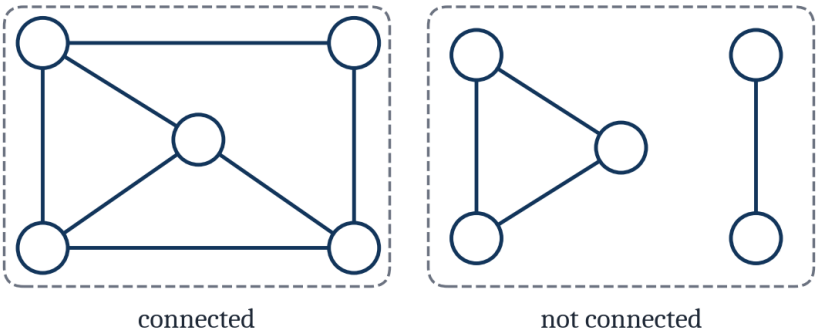


Figure 10.8 Connected and not connected. On the left, every vertex can be reached from every other. On the right the same vertices have been drawn with fewer edges, and the two vertices on the far right are cut off from the triangle on the left: no path joins them, so the graph is disconnected.

Our graph G is connected. To be sure of that you would in principle have to check all $\binom{5}{2} = 10$ pairs, but there is a shortcut we shall make into an algorithm in §10.9.1: pick any one vertex and see whether you can reach all the others from it. From a we have the paths a, $a\,b$, $a\,c$, $a\,d$ and $a\,b\,e$; that is all five vertices, so G is connected. The shortcut works because reachability in an undirected graph is *symmetric* — if you can get from a to u you can get from u to a by walking the same edges backward — and *transitive*: to get from u to v, go u to a and then a to v.

10.4.8 Connected components

DEFINITION. A **connected component** of an undirected graph is a maximal connected subgraph: a set of vertices, together with all the edges among them, that is connected, and to which no further vertex of the graph could be added without destroying connectedness.

The word **maximal** is doing all the work, so let us be very clear about it. In the right-hand picture of **Figure 10.8**, the single vertex at the top right forms a connected subgraph all by itself — one vertex, no edges, and trivially every pair of its vertices is joined by a path. But it is *not* a connected component, because we could add the vertex below it, and the edge between them, and still have something connected. A component is a piece you cannot grow.

Figure 10.9 is an example; the answer, which we shall confirm by program in §10.9.1, is three components.

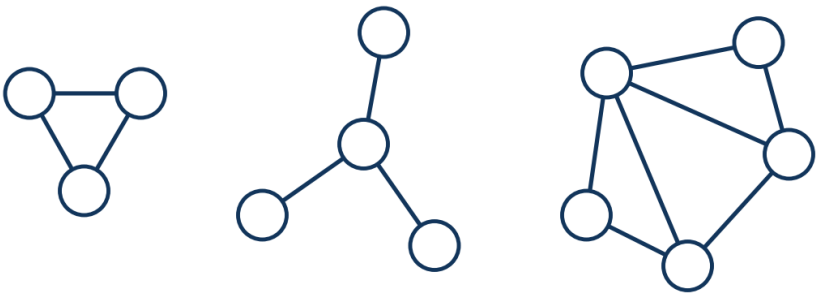


Figure 10.9 A graph with three connected components: a triangle on the left, a star in the middle, and a five-vertex piece on the right. No edge crosses between the three, and inside each one every vertex reaches every other.

Three facts about components, each of which we shall need:

1. **Every vertex lies in exactly one component.** Reachability splits V into groups: put u and v in the same group precisely when there is a path between them. Because reachability is reflexive, symmetric and transitive — an *equivalence relation* — the groups do not overlap and together they cover V. Each group, with its internal edges, is a component.
2. **A graph is connected exactly when it has one component.** That is the definition read from the other end.
3. **No edge joins two different components.** If an edge joined u in one component to v in another, then u could reach v, so they would be in the same group. This is why a search started inside a component can never escape it, which is the whole reason §10.9.1's algorithm works.

10.4.9 Strongly connected components, for directed graphs

In a directed graph, reachability is no longer symmetric — a one-way street goes one way — and so "connected" has to be said more carefully.

> **DEFINITION.** A directed graph is **strongly connected** if for every ordered pair of vertices u, v there is a directed path from u to v *and* a directed path from v to u. A **strongly connected component** is a maximal strongly connected subgraph.

Consider the three-arc directed graph $u \to v$, $v \to w$, $w \to u$. Every vertex reaches every other by going round the ring, so it is strongly connected: one strongly connected component containing all three. Now delete the arc $w \to u$. There is still a path $u \to v \to w$, but nothing at all leads back from w to u. The graph is no longer strongly connected, and it now has three strongly connected components, each a single vertex.

The directed graph of §10.8 — arcs $u \to v, u \to x, v \to y, w \to y, w \to z, x \to v, y \to x, z \to z$ — has four strongly connected components: $\{v, x, y\}$, because $v \to y \to x \to v$ is a directed ring; and the three singletons $\{u\}$, $\{w\}$ and $\{z\}$. Check $\{u\}$: u reaches v, but no arc enters u at all, so nothing reaches u, so u can share a component with nobody.

Computing strongly connected components takes *two* passes of depth-first search and is a classical application of the finishing times of §10.8.3. We shall not do it in this chapter, but everything it needs is built here.

10.4.10 Subgraphs and spanning subgraphs

> **DEFINITION.** A **subgraph** of $G = (V, E)$ is a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$, such that every edge in E' has both its endpoints in V'. A subgraph is **spanning** if $V' = V$ — that is, if it keeps every vertex and merely throws edges away.

A short form of the definition is *subgraph: subset of vertices and edges forming a graph*, and the clause "forming a graph" is the condition just stated: you may not keep an edge whose endpoint you have discarded, because then the edge would dangle.

Three worked examples on G.

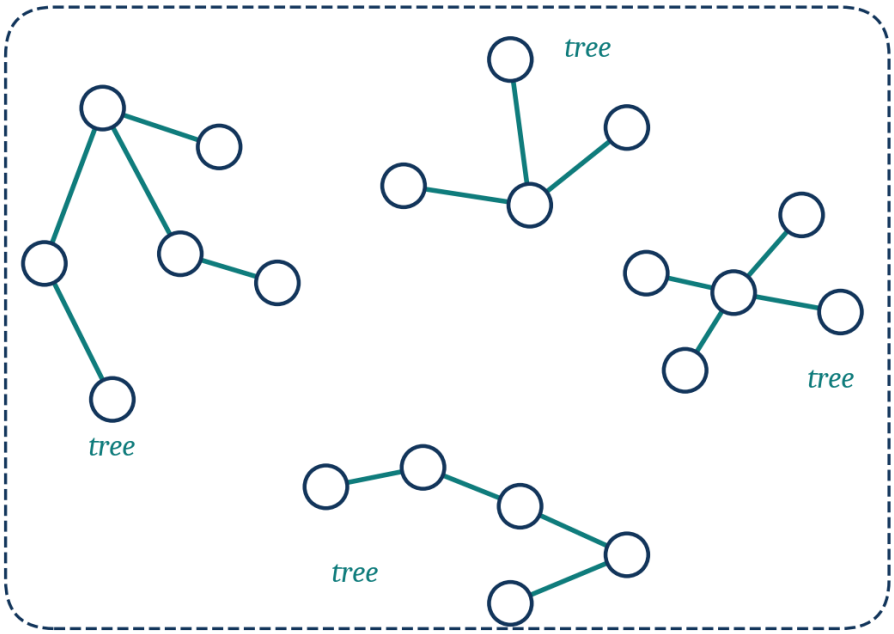
- $V' = \{a, b, e\}$, $E' = \{(a, b), (b, e)\}$. Legal: both endpoints of both edges survive. This is a subgraph, and it is a path.
- $V' = \{a, b, e\}$, $E' = \{(a, b), (b, e), (c, e)\}$. **Illegal**: the edge (c, e) needs c, and c has been discarded.
- $V' = \{a, b, c, d, e\}$, $E' = \{(a, b), (a, c), (a, d), (b, e)\}$. Legal, and spanning, because every vertex is present. Four of the seven edges have been thrown away.

Spanning subgraphs are the ones we care about, because both of this chapter's searches produce one. Breadth-first search from s returns a spanning subgraph of the component containing s (§10.7.10); depth-first search returns a spanning subgraph of the whole graph (§10.8.5). Chapter 11 is entirely about finding the *cheapest* spanning subgraph that is still connected — the minimum spanning tree.

10.4.11 Trees and forests

DEFINITION. A **(free) tree** is a connected acyclic undirected graph. A **forest** is an acyclic undirected graph — equivalently, a collection of trees, one per connected component.

In short: a *(free) tree* is a connected graph without cycles; a *forest* is a collection of trees. **Figure 10.10** illustrates both: four trees, and the dashed frame that gathers them into one forest.



forest: all four trees together

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.

The word **free** in "free tree" distinguishes this from the *rooted* trees of Chapter 6. A binary search tree has a root, and each node has a parent and up to two children; those roles are part of the data. A free tree has none of that. It is just a graph that happens to be connected and acyclic; no vertex is special. You may *choose* a vertex and call it the root, and then every other vertex acquires a parent — the neighbor one step closer to the root — and that is precisely the operation both of our searches perform.

Free trees have a beautiful rigidity, and the following three facts are used constantly. For an undirected graph G on $|V|$ vertices, any two of these three imply the third:

1. G is connected;
2. G is acyclic;
3. $|E| = |V| - 1$.

Let us at least see *why* a tree has exactly $|V| - 1$ edges, because the argument is short and the fact is used in §10.7.10. Induct on $|V|$. A tree with one vertex has no edges: $0 = 1 - 1$. Now take a tree T with $|V| \geq 2$ vertices. It must have a vertex of degree 1 — a **leaf**. (Why? Start anywhere and

keep walking along unused edges. You can never return to a vertex you have already visited, because that would close a cycle, and a tree has none. Since there are finitely many vertices, you must eventually reach a vertex with no unused edge leaving it, and that vertex has degree 1.) Delete that leaf and its one edge. What remains is still acyclic — deleting things cannot create a cycle — and still connected, because the leaf was on nobody's path between two other vertices. So it is a tree on $|V| - 1$ vertices, which by induction has $|V| - 2$ edges. Put the leaf back: $|V| - 1$ edges. ■

Two consequences to remember: a tree has exactly one simple path between any two of its vertices (if there were two different ones, joining them would make a cycle); and adding any single new edge to a tree creates exactly one cycle.

10.4.12 Complete graphs

> **DEFINITION.** A **complete graph** on n vertices, written K_n, is the simple undirected graph in which *every* pair of distinct vertices is adjacent.

How many edges does K_n have? Each of the n vertices is adjacent to the other $n - 1$, so the degrees sum to $n(n - 1)$; by the handshake lemma that sum is $2|E|$, so

$$|E| = \binom{n}{2} = \frac{n(n-1)}{2}. \tag{10.3}$$

For $n = 5$, Equation (10.3) gives 10. Our graph G has 7 of those 10 edges, so it is close to complete but not complete: (a, e), (b, c) and (b, d) are missing.

K_n matters because it is the *worst case for storage*. It tells us the largest $|E|$ can be for a given $|V|$, and therefore how far apart the three representations of §10.5 can possibly be.

10.4.13 Bipartite graphs

> **DEFINITION.** A graph is **bipartite** if its vertices can be split into two disjoint sets L and R — two *sides* — such that every edge has one endpoint in L and the other in R. Equivalently, no edge joins two vertices on the same side.

Bipartite graphs model every situation with two kinds of thing and a relation between the kinds: students and courses, workers and shifts, customers and orders, machines and jobs. "Every edge crosses" is exactly the statement that there is no relation *within* a kind.

Worked examples.

- A **path** p, q, r, s is bipartite: put p and r on the left, q and s on the right. Every edge — (p, q), (q, r), (r, s) — crosses.
- An **even cycle** p, q, r, s, p is bipartite: left $\{p, r\}$, right $\{q, s\}$.
- An **odd cycle** is not. Try a triangle p, q, r: put p on the left, then q must go right because of the edge (p, q), then r must go left because of (q, r) — and now the edge (r, p) joins two left vertices. No assignment works.
- Our graph G is **not** bipartite, and the triangle $a\,c\,d$ of **Figure 10.7** is the reason: any three mutually adjacent vertices cannot be split across two sides, because two of the three must land on the same side.
- $K_{2,3}$ — two vertices on the left, three on the right, all six crossing edges present — is bipartite by construction, and is a *complete* bipartite graph.

The characterization that makes bipartiteness testable is this, and §10.9.3 turns it into eleven lines of C++:

> **KEY IDEA.** A graph is bipartite **if and only if** it contains no cycle of odd length. The easy direction: walking round a cycle alternates sides at every step, so returning to your starting vertex requires an even number of steps. The other direction is the algorithm: try to 2-color the graph by breadth-first search, and either you succeed — which exhibits the two sides — or you find an edge between two vertices of the same color, which exhibits an odd cycle.

10.4.14 Weighted graphs

> **DEFINITION.** A **weighted graph** is a graph together with a function $w: E \to \mathbb{R}$ that assigns a number, the **weight**, to each edge.

The weight is whatever the application says it is: kilometers on a road, ohms on the resistor of **Figure 10.2**, minutes on a flight leg, rupiah on a delivery, capacity on a pipe.

Everything in *this* chapter is unweighted, and that is a deliberate choice rather than an oversight. When there are no weights, "shortest path" can only mean "fewest edges", and then a queue is enough: breadth-first search solves the problem in $O(V + E)$. Once edges carry weights, a queue is no longer enough — the route with fewest hops need not be the route with fewest kilometers — and you need a priority queue and Dijkstra's algorithm (Chapter 13), or, if some weights are negative, Bellman-Ford. It is worth being precise about this, because it is the most common source of wrong answers in practice:

> **WARNING.** Breadth-first search minimizes the **number of edges**. If your edges have lengths and the lengths are not all equal, BFS will confidently return a path that is not the shortest. The three-hop route may well be shorter in kilometers than the two-hop route. Use BFS when every edge costs the same, and Chapter 13 otherwise.

10.4.15 How many edges can there be? Sparse and dense

We now know enough to bound $|E|$ in terms of $|V|$, and the bound governs every design decision in §10.5. Write $n = |V|$ and $m = |E|$.

Undirected simple graphs. The most edges possible is that of the complete graph K_n, given by Equation (10.3):

$$0 \le m \le \binom{n}{2} = \frac{n(n-1)}{2} = O(n^2).$$

Directed graphs, no self-loops. Each ordered pair may carry an arc, so

$$0 \le m \le n(n-1) = O(n^2).$$

Directed graphs, self-loops allowed. Every one of the n^2 ordered pairs, including (v, v), may carry an arc:

$$0 \le m \le n^2.$$

So in every case $m = O(n^2)$, and the upper end of the range is quadratic. But there is also a *lower* bound worth knowing: if the graph is connected then $m \ge n - 1$, because a connected graph contains a spanning tree and a tree on n vertices has $n - 1$ edges (§10.4.11).

This spread — from about n to about n^2 — is what the words *sparse* and *dense* name.

DEFINITION. A graph is called **sparse** when $|E|$ is much closer to $|V|$ than to $|V|^2$ — informally, $|E| = O(|V|)$ or $O(|V|\log|V|)$. It is called **dense** when $|E|$ is close to its maximum, $\Theta(|V|^2)$.

The terms are informal but the distinction is real and is usually decided by the application, not by the algorithm. **Table 10.2** lays out the arithmetic for a few sizes, and the last column is the one that decides how you store the graph.

Table 10.2 How far apart sparse and dense really are. For a million vertices a matrix holds 10^{12} entries — a terabyte even at one byte each — while a sparse graph on the same vertices may have only a million edges. This single table is the reason §10.5 spends so long on representations.

$\|V\|$	$\|E\|$ If a tree (V−1)	$\|E\|$ If complete	Entries in a $V \times V$ matrix	Ratio, complete : tree
10	9	45	100	5
100	99	4 950	10 000	50
1 000	999	499 500	1 000 000	500
10 000	9 999	49 995 000	100 000 000	5 000
1 000 000	999 999	$\approx 5 \times 10^{11}$	10^{12}	$\approx 5 \times 10^5$

Which case is real? Almost all large graphs that arise in practice are sparse, and for a simple reason: real things have a bounded number of relationships. A road junction has at most a handful of roads; a person has hundreds of friends, not millions; a web page has tens of links, not billions. Dense graphs do occur — a distance matrix between every pair of cities, a similarity matrix between every pair of documents — but they are the exception, and they are small, precisely because n^2 grows so fast.

10.4.16 The whole vocabulary on one page

Table 10.3 collects every term of this section, with its meaning in one line and an instance in the chapter's graph G of **Figure 10.1**. Come back to it whenever a word in the rest of the chapter is unfamiliar.

Table 10.3 The complete vocabulary of §10.4, with an instance of each term in the graph of **Figure 10.1**. Every one of these words is used somewhere in the rest of the chapter.

Term	Meaning in one line	Instance in G
vertex	one of the things	a
edge	a joined pair of things	(a, b)
adjacent	joined by an edge	a and c are; a and e are not
incident	said of an edge and a vertex at its end	(b, e) is incident on b and on e
$\text{Adj}[v]$	the neighbors of v	$\text{Adj}[c] = \{a, d, e\}$
$\deg(v)$	number of edges at v	$\deg(b) = 2$
$\deg^+, \deg^-$	out-degree, in-degree (directed only)	—
path	sequence of vertices, consecutive ones adjacent	$a\ b\ e\ d\ c$
length of a path	number of edges in it	$a\ b\ e\ d\ c$ has length 4
simple path	path with no repeated vertex	$b\ e\ c$
$\delta(u, v)$	fewest edges on any path u to v	$\delta(b, c) = 2$
cycle	simple path whose last vertex is its first	$a\ c\ d\ a$
acyclic	having no cycle	G is not
connected	every pair joined by some path	G is
connected component	maximal connected subgraph	G has 1
strongly connected	directed: every pair joined both ways	directed graphs only
subgraph	a subset of vertices with a subset of the edges among them	$\{a, b, e\}$ with $(a, b), (b, e)$
spanning subgraph	subgraph keeping every vertex	$\{a, b, c, d, e\}$ with 4 of the 7 edges
tree	connected and acyclic	any spanning tree of G
forest	acyclic, possibly disconnected	a tree is a forest with one component
leaf	vertex of degree 1 in a tree	—
complete graph K_n	every pair adjacent	G is not: 3 pairs are missing
bipartite	vertices split into two sides, every edge crossing	G is not — it has a triangle
weighted	each edge carries a number	G is not
sparse / dense	$\lvert E\rvert$ near $\lvert V\rvert$ / near $\lvert V\rvert^2$	G has 7 of a possible 10: dense
source / sink	in-degree 0 / out-degree 0	directed graphs only
self-loop	edge from a vertex to itself	none in G

Table 10.3 (Continued) *The complete vocabulary of §10.4, with an instance of each term in the graph of Figure 10.1. Every one of these words is used somewhere in the rest of the chapter.*

Term	Meaning in one line	Instance in G
DAG	directed graph with no directed cycle	§10.9.5

10.5 Three Ways to Store a Graph

10.5.1 What a representation has to do

The question is simply this: *how can we represent a graph?* The first answer anyone gives is: *to start with, we can store the vertices and the edges in two containers, and store with each edge object*

references to its start and end vertices. **Figure 10.11** is the graph we use to make this concrete — a directed graph of flights, where each arc carries a flight number — and **Figure 10.12** shows the two-container structure built on it.

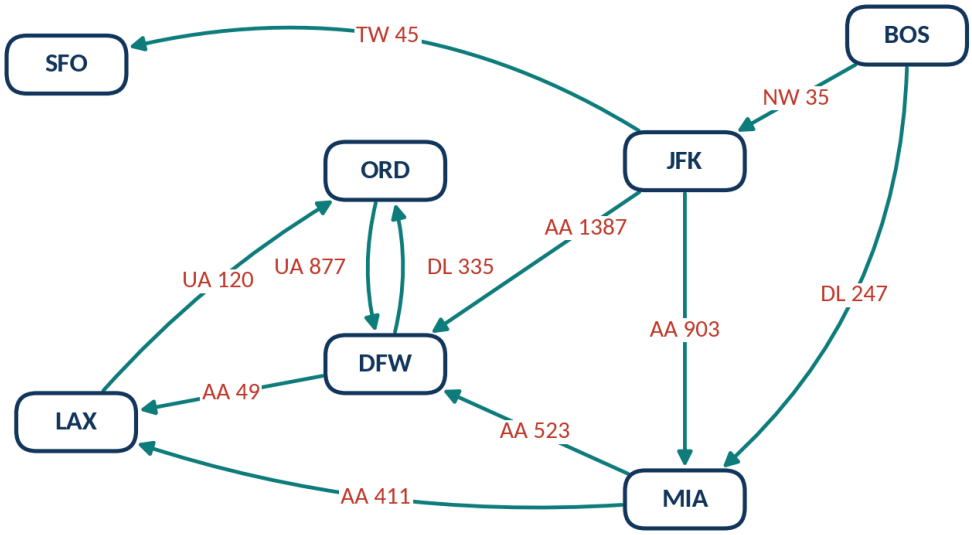


Figure 10.11 The running example for graph storage: a directed graph of flights between seven airports (BOS, JFK, ORD, SFO, LAX, DFW, MIA), each arc labeled with a flight number. There are eleven arcs. Because the arcs are directed, flight AA 49 from DFW to LAX is a different edge from flight DL 335 from DFW to ORD, and neither implies a flight in the reverse direction.

Before comparing representations we must agree on what we are comparing them *on*. Every algorithm in this chapter, and almost every graph algorithm anywhere, is built from just four questions:

Q1. Is there an edge from u to v? Call this the *adjacency test*. A matrix-flavored algorithm asks it constantly.

Q2. What is $\deg(v)$**?** Or, in a directed graph, $\deg^+(v)$.

Q3. List the neighbors of v. That is, enumerate $\mathrm{Adj}[v]$. **This is the one that matters most**: both searches in this chapter do nothing else, and doing it once per vertex is what makes them linear.

Q4. List all the edges. Needed by Kruskal's algorithm in Chapter 11, among others.

And one more measure, which is not a question but a constraint: **how much memory does the representation occupy?**

With those five yardsticks agreed, here are the three representations, each drawn in a figure.

10.5.2 The edge list

> **DEFINITION.** In the **edge-list** representation, the graph is stored as a container of vertex objects together with a container of edge objects, where each edge object holds a reference to its two endpoints.

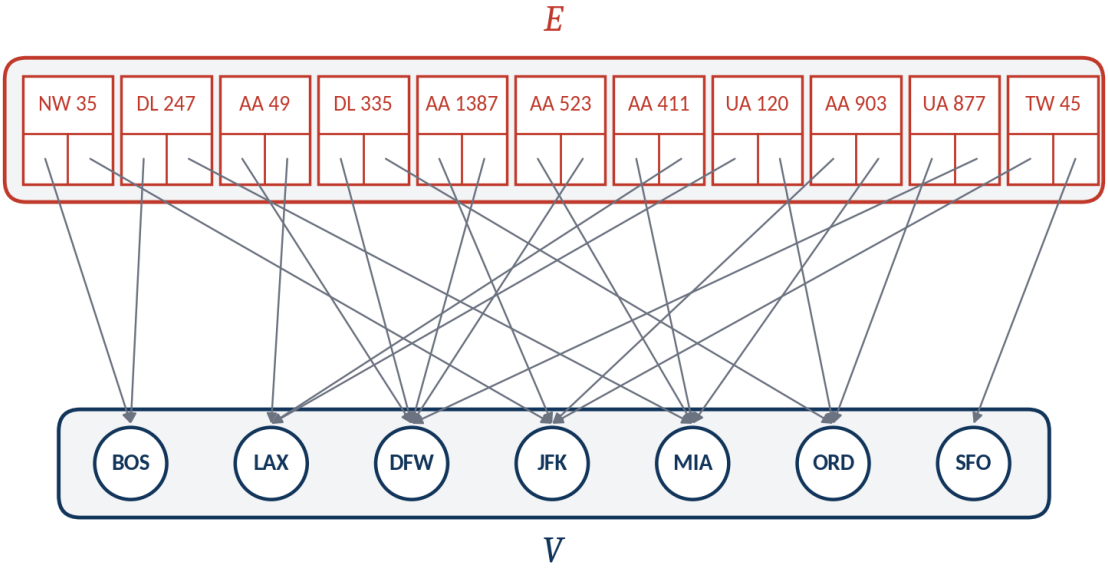


Figure 10.12 The edge list, built on the flight graph of **Figure 10.11**. The lower container holds the seven vertices; the upper container holds the eleven edges, and every edge object points down at its two endpoints. To find the flights leaving DFW you must inspect every one of the eleven edge objects, because nothing in the structure groups them by airport.

In C++ the whole representation is two lines:

```
int n;                          // the vertices are 0, 1, ..., n-1
std::vector<std::pair<int,int>> E;   // one entry per edge
```

For the graph of **Figure 10.1** the container holds

$$E = [(a,b),\ (a,c),\ (a,d),\ (b,e),\ (c,d),\ (c,e),\ (d,e)].$$

Space. Two vertex references per edge, plus the vertices themselves: $\Theta(V+E)$. With integers standing in for references that is $2|E|$ machine words for the edges — for our graph, 14.

Q1, adjacency test. There is nothing to do but scan the whole container looking for the pair. $\Theta(E)$ in the worst case.

Q2, degree. Scan the whole container counting the entries that mention v. $\Theta(E)$.

Q3, list the neighbors. Scan the whole container collecting the entries that mention v. $\Theta(E)$ — and this is the representation's real flaw: *finding the edges incident on a given vertex is inefficient, since it requires examining the entire edge sequence.*

Q4, list all the edges. Read the container out. $\Theta(E)$, which is optimal — there is no faster way to print $|E|$ things.

So the edge list is the best of the three at exactly one job, Q4, and the worst at the other three. That makes it far from useless: Kruskal's algorithm for minimum spanning trees (Chapter 11) begins by sorting *all* the edges and then never asks Q1, Q2 or Q3, so an edge list is precisely the right structure for it. It is also, in practice, the format graphs arrive in — a file with one edge per line — so real programs usually read an edge list and convert it to something else. That conversion is nine lines and appears in **Listing 10.1**.

10.5.3 The adjacency list

DEFINITION. The **adjacency list** of a vertex v is a sequence holding the vertices adjacent to v. In the **adjacency-list representation**, the graph is stored as an array of $|V|$ such sequences, one per vertex.

In short: *the adjacency list of a vertex v is a sequence of the vertices adjacent to v; the graph is represented by the adjacency lists of all its vertices.* **Figure 10.13** draws it for the chapter's graph.

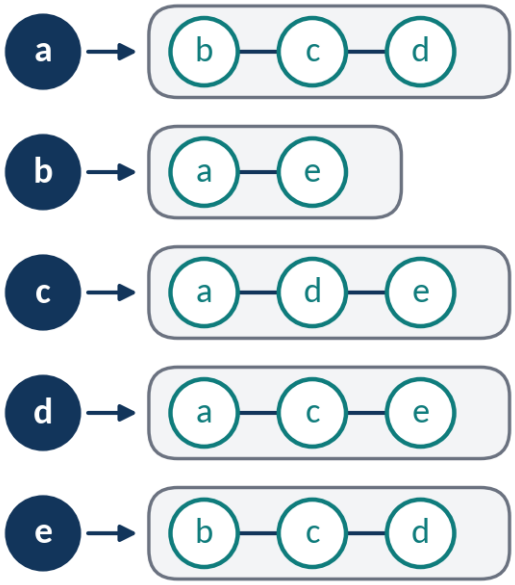


Figure 10.13 The adjacency lists of the chapter's graph. Five lists, one per vertex. Vertex a's list holds b, c, d; vertex b's holds a, e; and so on. The lists together hold 14 entries, which is $2|E|$ — the handshake lemma, made visible.

In C++:

```
int n;
std::vector<std::vector<int>> adj;   // adj[v] holds the neighbors of v
```

For our graph the five lists are

$$\mathrm{Adj}[a] = [b, c, d], \quad \mathrm{Adj}[b] = [a, e], \quad \mathrm{Adj}[c] = [a, d, e], \quad \mathrm{Adj}[d] = [a, c, e], \quad \mathrm{Adj}[e] = [b, c, d].$$

Note carefully that in an **undirected** graph every edge appears **twice**: the edge (a, b) appears as b in $\mathrm{Adj}[a]$ and again as a in $\mathrm{Adj}[b]$. That is not redundancy to be optimized away; it is what makes Q3 fast from either end. In a **directed** graph each arc appears once, in the list of its tail.

Space. $|V|$ list headers plus one entry per directed edge-slot. Undirected: $\Theta(V + 2E) = \Theta(V + E)$. Directed: $\Theta(V + E)$. For our graph: 5 headers plus 14 entries, 19 words.

Q1, adjacency test. Walk $\mathrm{Adj}[u]$ looking for v: $O(\deg(u))$. That is fast when degrees are small and slow when they are not — in the worst case $O(V)$.

Q2, degree. The length of the list, which a `std::vector` knows: $\Theta(1)$.

Q3, list the neighbors. Read $\mathrm{Adj}[v]$: $\Theta(\deg(v))$, which is optimal, since you cannot list $\deg(v)$ things faster.

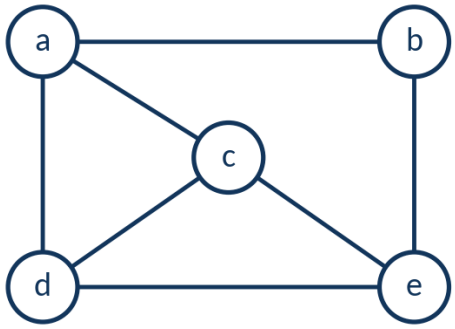
Q4, list all the edges. Walk every list: $\Theta(V + E)$. Slightly worse than the edge list's $\Theta(E)$, because you must visit the $|V|$ headers even if some lists are empty, and each undirected edge is met twice so you must remember to emit it once.

> **KEY IDEA.** The adjacency list is the default. It is the representation both of this chapter's searches assume — both BFS and DFS are analyzed for an adjacency-list representation — because Q3 is the only question a search asks, and the adjacency list answers Q3 optimally while using only $\Theta(V + E)$ space. Unless you have a specific reason to do otherwise, store your graph this way.

10.5.4 The adjacency matrix

> **DEFINITION.** The **adjacency matrix** of a graph on n vertices is an $n \times n$ array M in which $M[i][j]$ is **true** if there is an edge from vertex i to vertex j, and **false** if there is not.

Figure 10.14 pictures it: the graph on the left, the matrix of T's and F's on the right.



	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.

Writing the matrix out with 1 for true and 0 for false, and reading **Figure 10.14** row by row:

$$M = \begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 1 & 0 \\ b & 1 & 0 & 0 & 0 & 1 \\ c & 1 & 0 & 0 & 1 & 1 \\ d & 1 & 0 & 1 & 0 & 1 \\ e & 0 & 1 & 1 & 1 & 0 \end{array}$$

Check one row against E: row c has 1 in columns a, d, e, and $(a, c), (c, d), (c, e)$ are indeed the three edges at c. Check one absence: $M[a][e] = 0$, and $(a, e) \notin E$. Good.

Three structural facts, each with a reason:

- **The matrix is symmetric** — $M[i][j] = M[j][i]$ — precisely when the graph is undirected, because then the edge (i, j) and the edge (j, i) are the same edge. For a directed graph the matrix is in general not symmetric, and the asymmetry *is* the direction information.
- **The diagonal is all zero** exactly when there are no self-loops. A self-loop at v puts a 1 at $M[v][v]$.
- **The row sum is the degree.** In row v, the number of 1s is the number of neighbors of v. In a directed graph the row sum is $\deg^+(v)$ and the column sum is $\deg^-(v)$.

Space. $\Theta(V^2)$, always — 25 words for our five-vertex graph, whatever $|E|$ is — $O(n^2)$ for n vertices. This is the representation's whole problem: a graph with a million vertices needs 10^{12} entries, which is a terabyte at one byte apiece and simply cannot be allocated, no matter how few edges the graph has.

Q1, adjacency test. One array lookup: $\Theta(1)$. This is the representation's whole virtue, and it is a real one.

Q2, degree. Sum a row: $\Theta(V)$.

Q3, list the neighbors. Walk a row and collect the 1s: $\Theta(V)$ — and note that this is $\Theta(V)$ *even when v has no neighbors at all*, because you must still inspect $|V|$ cells to find that out. That is exactly why a matrix-based search costs $\Theta(V^2)$ rather than $O(V + E)$, and §10.9.8 measures the difference.

Q4, list all the edges. Scan the whole matrix: $\Theta(V^2)$.

A last remark on implementation. `std::vector<bool>` in C++ is a bit-packed specialization, which makes a matrix eight times smaller at the price of slower element access and some surprising behavior (its `operator[]` does not return a real `bool&`). For clarity the program below uses `std::vector<char>`; for a genuinely large dense matrix, bit-packing is the right trade and `std::vector<bool>` or a hand-rolled bitset is the tool.

10.5.5 Space, side by side

Table 10.4 Space cost of the three representations. For the tiny graph of **Figure 10.1** the three are within a factor of two of each other, which is why small examples mislead. At a million vertices the matrix is off the table entirely.

Representation	Space, in general	Space for (V=5, E=7)	Space for V=10^6, E=4×10^6
Edge list	$\Theta(V + E)$	14 words	$\approx 8 \times 10^6$ words
Adjacency list	$\Theta(V + E)$	19 words	$\approx 9 \times 10^6$ words
Adjacency matrix	$\Theta(V^2)$	25 words	10^{12} entries — impossible

The lesson of **Table 10.4** is that small examples hide the difference. On five vertices the matrix costs 25 words against the adjacency list's 19 — nobody would care. The break-even point is where $V^2 \approx V + 2E$, that is roughly $E \approx V^2/2$: the matrix wins on space only for graphs that are genuinely near-complete.

10.5.6 The four questions, side by side

Table 10.5 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 $\mathrm{Adj}[v]$	$\Theta(E)$	$\Theta(\deg v)$	$\Theta(V)$
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	—	$\Theta(V + E)$	$\Theta(V^2)$

10.5.7 Choosing

The choice follows from two questions about your application.

First: is the graph sparse or dense? If $|E|$ is much less than $|V|^2$ — which is the usual case — the matrix wastes a great deal of memory, and worse, it makes every traversal quadratic. Use an adjacency list. If the graph is genuinely dense, the matrix costs no more memory than the lists and is faster at everything except Q4.

Second: which of Q1 and Q3 dominates? If your algorithm's inner loop is "is (u, v) an edge?", a matrix's $\Theta(1)$ answer is worth paying for; the Floyd-Warshall algorithm of Chapter 15 is built this way and uses a matrix without apology. If the inner loop is "give me the neighbors of v" — which is every traversal, every search, Prim, Dijkstra, BFS, DFS — use an adjacency list.

Table 10.6 When to reach for which representation, and the reason in each case. If in doubt, choose the adjacency list; it is the right answer far more often than not.

Situation	Use	Why
A search, a traversal, BFS, DFS, Prim, Dijkstra	Adjacency list	Q3 is the only question asked, and the list answers it optimally
Sparse graph, $E = O(V)$ or $O(V \log V)$	Adjacency list	$\Theta(V + E)$ space rather than $\Theta(V^2)$
Dense graph, $E = \Theta(V^2)$, with constant-time edge tests needed	Adjacency matrix	$\Theta(V^2)$ space costs nothing extra, and Q1 becomes $\Theta(1)$
All-pairs shortest paths, transitive closure, matrix-style algorithms	Adjacency matrix	The algorithm *is* matrix arithmetic (Chapter 15)
Kruskal's algorithm; reading a graph from a file	Edge list	The only question asked is Q4, and the edge list answers it optimally
Edges are inserted and deleted constantly, with edge tests in between	Adjacency matrix, or adjacency list of hash sets	$\Theta(1)$ insert, delete and test

10.5.8 All three, in one program

Listing 10.1 builds the graph of **Figure 10.1** three times over, in all three representations, and answers all four questions from each. Running the three side by side is the fastest way to convince yourself that they hold the same information — the degrees must agree, the adjacency tests must agree — and that they cost different amounts to consult.

Listing 10.1 The chapter's graph stored three ways at once, with all four basic queries answered from each representation. Compile with `g++ -std=c++17 -O2`.

```
// Three representations of one graph, side by side, on the graph of Chapter 10:
//   V = {a,b,c,d,e},  E = {(a,b),(a,c),(a,d),(b,e),(c,d),(c,e),(d,e)}
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

// ---------- Representation 1: the edge list ----------
struct EdgeList {
    int n;                                  // number of vertices
    vector<pair<int,int>> E;                // one entry per edge
    explicit EdgeList(int n_) : n(n_) {}
    void addEdge(int u, int v) { E.push_back({u, v}); }

    bool areAdjacent(int u, int v) const {          // Theta(E)
        for (const auto& e : E)
            if ((e.first == u && e.second == v) || (e.first == v && e.second == u))
                return true;
        return false;
    }
    int degree(int u) const {                       // Theta(E)
        int d = 0;
        for (const auto& e : E) if (e.first == u || e.second == u) ++d;
        return d;
    }
    vector<int> neighbors(int u) const {           // Theta(E)
        vector<int> out;
        for (const auto& e : E) {
            if (e.first  == u) out.push_back(e.second);
            if (e.second == u) out.push_back(e.first);
        }
        sort(out.begin(), out.end());
        return out;
    }
    size_t words() const { return 2 * E.size(); }   // two ints stored per edge
};

// ---------- Representation 2: the adjacency list ----------
struct AdjList {
    int n;
    vector<vector<int>> adj;
    explicit AdjList(int n_) : n(n_), adj(n_) {}
    void addEdge(int u, int v) { adj[u].push_back(v); adj[v].push_back(u); }
```

Listing 10.1 (Continued) *The chapter's graph stored three ways at once, with all four basic queries answered from each representation. Compile with g++ -std=c++17 -O2.*

```
    bool areAdjacent(int u, int v) const {          // O(deg(u))
        for (int w : adj[u]) if (w == v) return true;
        return false;
    }
    int degree(int u) const { return (int)adj[u].size(); }       // O(1)
    const vector<int>& neighbors(int u) const { return adj[u]; } // O(1) to reach
    size_t words() const {                          // n heads + 2E entries
        size_t t = n;
        for (const auto& l : adj) t += l.size();
        return t;
    }
};

// ---------- Representation 3: the adjacency matrix ----------
struct AdjMatrix {
    int n;
    vector<vector<char>> M;                 // char, not bool: predictable layout
    explicit AdjMatrix(int n_) : n(n_), M(n_, vector<char>(n_, 0)) {}
    void addEdge(int u, int v) { M[u][v] = 1; M[v][u] = 1; }

    bool areAdjacent(int u, int v) const { return M[u][v] != 0; }  // Theta(1)
    int degree(int u) const {                                       // Theta(V)
        int d = 0;
        for (int v = 0; v < n; ++v) if (M[u][v]) ++d;
        return d;
    }
    vector<int> neighbors(int u) const {                          // Theta(V)
        vector<int> out;
        for (int v = 0; v < n; ++v) if (M[u][v]) out.push_back(v);
        return out;
    }
    size_t words() const { return (size_t)n * n; }
};

static const string NAME = "abcde";
static void printList(const vector<int>& v) {
    cout << "{";
    for (size_t i = 0; i < v.size(); ++i) cout << (i ? "," : "") << NAME[v[i]];
    cout << "}";
}

int main() {
    const int n = 5;
    const vector<pair<int,int>> edges = {{0,1},{0,2},{0,3},{1,4},{2,3},{2,4},{3,4}};
```

Listing 10.1 (Continued) *The chapter's graph stored three ways at once, with all four basic queries answered from each representation. Compile with g++ -std=c++17 -O2.*

```
    EdgeList  el(n);
    AdjList   al(n);
    AdjMatrix am(n);
    for (auto e : edges) { el.addEdge(e.first,e.second); al.addEdge(e.first,e.second);
am.addEdge(e.first,e.second); }

    cout << "|V| = " << n << ", |E| = " << edges.size() << "\n\n";

    cout << "adjacency matrix (1 = edge present)\n    ";
    for (int j = 0; j < n; ++j) cout << NAME[j] << ' ';
    cout << "\n";
    for (int i = 0; i < n; ++i) {
        cout << "  " << NAME[i] << ' ';
        for (int j = 0; j < n; ++j) cout << (am.areAdjacent(i,j) ? '1' : '0') << ' ';
        cout << "\n";
    }

    cout << "\nadjacency lists\n";
    for (int u = 0; u < n; ++u) {
        cout << "  " << NAME[u] << " -> "; printList(al.neighbors(u)); cout << "\n";
    }

    cout << "\nedge list\n  ";
    for (auto e : el.E) cout << "(" << NAME[e.first] << "," << NAME[e.second] << ") ";
    cout << "\n";

    cout << "\ndegrees (all three representations must agree)\n";
    int sum = 0;
    for (int u = 0; u < n; ++u) {
        int a = el.degree(u), b = al.degree(u), c = am.degree(u);
        cout << "  deg(" << NAME[u] << ") = " << a << " " << b << " " << c
             << (a==b && b==c ? "   agree" : "   MISMATCH") << "\n";
        sum += b;
    }
    cout << "  sum of degrees = " << sum << " = 2|E| = " << 2*edges.size() << "\n";

    cout << "\nadjacency queries\n";
    const int pairs[4][2] = {{0,1},{0,4},{2,4},{1,3}};
    for (auto& p : pairs)
        cout << "  " << NAME[p[0]] << "~" << NAME[p[1]] << " : "
             << (el.areAdjacent(p[0],p[1]) ? "yes" : "no ") << " "
             << (al.areAdjacent(p[0],p[1]) ? "yes" : "no ") << " "
             << (am.areAdjacent(p[0],p[1]) ? "yes" : "no ") << "\n";
```

Listing 10.1 (Continued) *The chapter's graph stored three ways at once, with all four basic queries answered from each representation. Compile with g++ -std=c++17 -O2.*

```
    cout << "\nstorage, counted in machine words\n";
    cout << "  edge list       " << el.words() << "\n";
    cout << "  adjacency list  " << al.words() << "\n";
    cout << "  adjacency matrix " << am.words() << "\n";
    return 0;
}
```

Its output:

```
|V| = 5, |E| = 7

adjacency matrix (1 = edge present)
    a b c d e
  a 0 1 1 1 0
  b 1 0 0 0 1
  c 1 0 0 1 1
  d 1 0 1 0 1
  e 0 1 1 1 0

adjacency lists
  a -> {b,c,d}
  b -> {a,e}
  c -> {a,d,e}
  d -> {a,c,e}
  e -> {b,c,d}

edge list
  (a,b) (a,c) (a,d) (b,e) (c,d) (c,e) (d,e)

degrees (all three representations must agree)
  deg(a) = 3 3 3   agree
  deg(b) = 2 2 2   agree
  deg(c) = 3 3 3   agree
  deg(d) = 3 3 3   agree
  deg(e) = 3 3 3   agree
  sum of degrees = 14 = 2|E| = 14

adjacency queries
  a~b : yes yes yes
  a~e : no  no  no
  c~e : yes yes yes
  b~d : no  no  no

storage, counted in machine words
  edge list       14
  adjacency list  19
  adjacency matrix 25
```

Four things to notice in that output, because each confirms something we proved by hand.

The printed matrix is exactly the matrix of **Figure 10.14** with 1 for T and 0 for F. The printed adjacency lists are exactly the five lists of **Figure 10.13**. The degrees agree across all three representations and sum to $14 = 2|E|$, which is the handshake lemma of §10.4.3 confirmed by machine. And the three storage figures, 14, 19 and 25, are the middle column of **Table 10.4**.

10.6 Searching a Graph

Everything so far has been description. Now we compute.

The task can be stated in one line: *a systematic search of every edge and vertex of the graph*. Two words in that line carry the weight.

Systematic means: by a rule, not by luck. The rule must guarantee that when the search stops, every vertex it should have visited has been visited, and no vertex has been visited twice. Visiting twice is not merely wasteful; in a graph with a cycle it is fatal, because the second visit leads to a third and the program never stops.

Every edge and vertex means the search must be exhaustive over the part of the graph it can reach — and note that "can reach" is a real restriction. A search started at s visits exactly the

vertices reachable from s, which in a disconnected graph is not all of them. That is a feature, and §10.9.1 makes an algorithm out of it.

There are exactly two natural rules, and they differ in one decision: **when you have several unexplored places to go, which do you go to first?**

- Go to the one you learned about **earliest**. That is a **queue** — first in, first out — and the result is **breadth-first search**.
- Go to the one you learned about **most recently**. That is a **stack** — last in, first out — and the result is **depth-first search**.

That is the whole difference. Two algorithms, identical but for the container in the middle, and yet they compute quite different things: one finds shortest paths, the other finds the structure of cycles and precedences. Hold that sentence in mind through the next two sections; it will keep you oriented.

Both algorithms need a way to remember what has already been seen, and both use the same device: a **color** on each vertex.

> **DEFINITION.** During a search, every vertex is exactly one of three colors. **White** means *undiscovered*: the search has not yet reached it. **Gray** means *discovered but not finished*: the search has reached it, but has not yet looked at all of its edges. **Black** means *finished*: the search has reached it and has examined every edge leaving it.

Every vertex starts white, turns gray exactly once, and turns black exactly once; the colors only ever move white → gray → black, never backward. That one-way property is what guarantees termination, and it is worth saying why in advance of either algorithm: a vertex is put into the queue or stack only at the moment it turns gray, it turns gray only once, and so it enters the container at most once. There are $|V|$ vertices, so there are at most $|V|$ container insertions, so the main loop runs at most $|V|$ times. No loop, no hang.

> **KEY IDEA.** The gray vertices are the **frontier** of the search: the boundary between the part of the graph already explored and the part not yet seen. Breadth-first search and depth-first search are two different disciplines for managing that frontier. BFS keeps it as a queue and so the frontier spreads outward in rings of equal distance. DFS keeps it as a stack and so the frontier is a single long chain plunging away from the start: *the gray vertices form a linear chain*, as §10.8.1 will show.

A note on the color names. This book writes *white*, *gray*, *black*, following Cormen, Leiserson, Rivest and Stein [1]. Other books say *unvisited*, *visiting*, *visited*, or *undiscovered*, *in progress*, *done*. They are the same three states. And if you only need to know *whether* a vertex has been seen, two states suffice — a plain `bool visited[]` — which is why much competitive-programming code uses a boolean array and no colors at all. The three-color scheme earns its extra state in §10.8, where the difference between gray and black is exactly what distinguishes a back edge from a cross edge, and hence exactly what detects a cycle.

10.7 Breadth-First Search

10.7.1 The labyrinth and the string

The classic picture for breadth-first search, from Goodrich and Tamassia [2], is worth quoting in full, because it is better than the algorithm at explaining the algorithm.

BFS in an undirected graph G is like wandering in a labyrinth with a string. The starting vertex s, it is assigned a distance 0. In the first round, the string is unrolled the length of one edge, and all of the edges that are only one edge away from the anchor are visited (discovered), and assigned distances of 1. In the second round, all the new edges that can be reached by unrolling the string 2 edges are visited and assigned a distance of 2. This continues until every vertex has been assigned a level. The label of any vertex v corresponds to the length of the shortest path (in terms of edges) from s to v.

Take the metaphor seriously. You are standing at s, holding the end of a ball of string, and you have tied the other end to s. You are not allowed to run off down a corridor; you must unroll the string **one meter at a time**, and at each length you must walk the whole circle of places that meter reaches before you unroll any further. After one meter you have stood in every room one corridor from s. After two meters, every room two corridors away — and crucially, *only* rooms you had not already stood in, because a room you reached at one meter is not two meters away just because there is also a longer way to it.

That last clause is the whole reason the algorithm is correct, and it is also the reason it needs the colors. The moment a room is reached, its distance is fixed forever, because it was reached at the earliest possible unrolling. Any later route to it must be at least as long. So:

> **KEY IDEA.** In breadth-first search a vertex is discovered at the *earliest* moment it can be discovered, and its distance is written down then and never revised. This "first touch wins" property is what makes BFS a shortest-path algorithm, and it is exactly what fails if you forget to check the color before enqueueing.

The search proceeds in **rings**: first s alone, then everything at distance 1, then everything at distance 2, and so on. The colors make the rings visible. When the search has finished with distance k and is about to start on distance $k+1$, the black vertices are those at distance $\leq k-1$, the gray vertices are exactly the ring at distance k, and everything at distance $>k$ is still white.

10.7.2 The three things BFS computes

Breadth-first search from a source vertex s maintains three arrays, and produces all three as its answer.

`color[v]` — white, gray or black, as in §10.6. This is bookkeeping; it is thrown away at the end. Its only job is to ensure no vertex is discovered twice.

`d[v]` — the **distance** from s to v: the number of edges on the shortest path. It starts at ∞ for every vertex but s, meaning "not known to be reachable", and is set exactly once, at the moment v is discovered. The symbol ∞ is used rather than -1 because it makes the arithmetic honest: it really is the case that no path of finite length has yet been found. In a program, `INT_MAX` or any value larger than $|V|$ will do, since no shortest path exceeds $|V|-1$ edges (§10.4.5).

`π[v]` — the **predecessor** of v, pronounced "pi of v": the vertex from which v was discovered. It starts at NIL for everybody and is set exactly once, at the same moment `d[v]` is. The π array is what lets you recover the actual *route*, not just its length: to go from s to v, look up $\pi[v]$, then $\pi[\pi[v]]$, and keep going until you reach s; then reverse what you have written down. §10.7.10 shows that the π array is a tree, and §10.7.11 proves that the path it encodes is a shortest one.

10.7.3 The algorithm

Figure 10.15 gives BFS, color-coded by what each block of lines does.

```
BFS (G, s)
01  for each vertex u ∈ V[G] − {s}
02      color[u] ← WHITE
03      d[u] ← ∞
04      π[u] ← NIL
05  color[s] ← GRAY
06  d[s] ← 0
07  π[s] ← NIL
08  Q ← {s}
09  while Q ≠ ∅ do
10      u ← head[Q]
11      for each v ∈ Adj[u] do
12          if color[v] = WHITE then
13              color[v] ← GRAY
14              d[v] ← d[u] + 1
15              π[v] ← u
16              ENQUEUE (Q, v)
17      DEQUEUE (Q)
18      color[u] ← BLACK
```

Initialize every vertex except s

Initialize the source and the queue

Handle all of u's children before handling any children of children

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.

Let us read it line by line, in words.

- **Lines 01–04.** For every vertex u other than the source: paint it white, set its distance to ∞, and set its predecessor to NIL. Nothing is known about anything yet. This loop runs $|V| - 1$ times and does constant work each time, so it costs $\Theta(V)$.
- **Line 05.** Paint the source gray. It has been discovered — we are standing on it — but we have not yet looked at its edges.
- **Line 06.** The source's distance to itself is 0.
- **Line 07.** The source has no predecessor; nothing discovered it.
- **Line 08.** Put the source in the queue. The queue now holds the entire frontier, which is one vertex.
- **Line 09.** While the queue is not empty — that is, while there is still frontier to work on.
- **Line 10.** Let u be the vertex at the head of the queue. Because it is a *queue*, this is the vertex that has been waiting longest, hence the one discovered earliest, hence one of the closest to s that is still unfinished.
- **Line 11.** For every neighbor v of u — that is, walk Adj$[u]$ once.
- **Line 12.** If v is white, it has never been seen before. This single test is what prevents the algorithm from looping forever, and it is what makes "first touch wins" true.
- **Line 13.** Paint v gray: it is now discovered and joins the frontier.
- **Line 14.** Its distance is one more than u's. We reached v by taking one extra edge from u.

- **Line 15.** Its predecessor is u.
- **Line 16.** Put v at the *back* of the queue. It will be dealt with after everything already waiting — which is everything closer to s.
- **Line 17.** Remove u from the queue; we are done with it.
- **Line 18.** Paint u black. Every edge leaving u has now been examined.

> **WARNING — two slips to watch for, and what they teach.** Two lines of **Figure 10.15** are easy to get wrong when the algorithm is copied by hand. Line 07 must read $\pi[s] \leftarrow \text{NIL}$, not $\pi[u] \leftarrow \text{NIL}$, and line 15 must read $\pi[v] \leftarrow u$, not $\pi[u] \leftarrow u$ — we are recording that v was discovered from u, so it is v's predecessor that changes, not u's. The reason to dwell on them is that the second slip is the single most common real bug in hand-written BFS code. If you write $\pi[u] \leftarrow u$, the program still terminates, still visits every vertex, and still computes every distance correctly — and then returns nonsense when you ask it for a path. A bug that corrupts only the part of the answer you did not immediately look at is the most expensive kind there is.

Set in the notation of this book, the algorithm is **Pseudocode 10.1**. Its line numbers match those of **Figure 10.15**, so the line-by-line reading above applies to it unchanged.

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
```

10.7.4 A worked example, one step at a time

The graph below is the one to learn on. Its vertices are laid out in two rows of four:

$$V = \{r, s, t, u, v, w, x, y\}, \qquad E = \{(r,s),\ (r,v),\ (s,w),\ (t,u),\ (t,w),\ (t,x),\ (u,y),\ (w,x),\ (x,y)\}.$$

Eight vertices, nine edges. As a check on our reading of the picture, let us confirm the handshake lemma: $\deg(r) = 2$, $\deg(s) = 2$, $\deg(t) = 3$, $\deg(u) = 2$, $\deg(v) = 1$, $\deg(w) = 3$, $\deg(x) = 3$, $\deg(y) = 2$; total $2 + 2 + 3 + 2 + 1 + 3 + 3 + 2 = 18 = 2 \times 9$. Correct.

The source is s. One further thing must be fixed before a trace can be reproduced: **the order in which each adjacency list is stored**, because that order decides which of two equally-close

vertices is discovered first. Reading the queue contents off **Figure 10.16** to **Figure 10.19** tells us the order, and it is:

$$\begin{aligned}\mathrm{Adj}[s] = [w, r],\ \mathrm{Adj}[r] = [s, v],\ \mathrm{Adj}[w] = [s, t, x],\ \mathrm{Adj}[t] \\ = [u, w, x],\ \mathrm{Adj}[x] = [t, w, y],\ \mathrm{Adj}[v] = [r],\ \mathrm{Adj}[u] = [t, y],\ \mathrm{Adj}[y] \\ = [u, x].\end{aligned}$$

> **WARNING.** The *distances* a BFS computes do not depend on the order of the adjacency lists — they are a property of the graph. The *tree*, the π array, and the order of the queue all do. So if your trace disagrees with a friend's on which vertex was discovered first, you may both be right; if you disagree on a distance, at least one of you is wrong.

Now the trace. Each numbered step is one turn of the `while` loop: take the head of the queue, walk its adjacency list, then blacken it. The queue is written left-to-right with the head on the left, and each vertex is shown with its distance.

Initialization. Every vertex white with $d = \infty$, except s: gray, $d[s] = 0$, $\pi[s] =$ NIL. Enqueue s.

$$Q = (s: 0)$$

Step 1. Dequeue s**.** $\mathrm{Adj}[s] = [w, r]$. w is white: paint gray, $d[w] = d[s] + 1 = 1$, $\pi[w] = s$, enqueue. $Q = (w: 1)$. r is white: paint gray, $d[r] = 1$, $\pi[r] = s$, enqueue. $Q = (w: 1,\ r: 1)$. Blacken s.

$$Q = (w: 1,\ r: 1)$$

Step 2. Dequeue w**.** $\mathrm{Adj}[w] = [s, t, x]$. s is black — skip it. (This is the algorithm noticing it has been here before.) t is white: gray, $d[t] = d[w] + 1 = 2$, $\pi[t] = w$, enqueue. x is white: gray, $d[x] = 2$, $\pi[x] = w$, enqueue. Blacken w.

$$Q = (r: 1,\ t: 2,\ x: 2)$$

Step 3. Dequeue r**.** $\mathrm{Adj}[r] = [s, v]$. s is black — skip. v is white: gray, $d[v] = d[r] + 1 = 2$, $\pi[v] = r$, enqueue. Blacken r.

$$Q = (t: 2,\ x: 2,\ v: 2)$$

Pause here and look at the queue. It holds three vertices, all at distance 2. The ring at distance 1 — w and r — is now entirely black, and the whole of the ring at distance 2 is gray and waiting. The rings are real.

Step 4. Dequeue t**.** $\mathrm{Adj}[t] = [u, w, x]$. u is white: gray, $d[u] = d[t] + 1 = 3$, $\pi[u] = t$, enqueue. w is black — skip. x is **gray** — skip. Note the distinction: x was discovered at step 2 from w, and it is still sitting in the queue. There is an edge (t, x), so we could have reached x from t and written $d[x] = d[t] + 1 = 3$. We do not, because x already has a smaller distance. First touch wins. Blacken t.

$$Q = (x: 2,\ v: 2,\ u: 3)$$

Step 5. Dequeue x**.** $\mathrm{Adj}[x] = [t, w, y]$. t is black — skip. w is black — skip. y is white: gray, $d[y] = d[x] + 1 = 3$, $\pi[y] = x$, enqueue. Blacken x.

$$Q = (v{:}\,2,\ u{:}\,3,\ y{:}\,3)$$

Step 6. Dequeue v**.** $\mathrm{Adj}[v] = [r]$. r is black — skip. Nothing discovered; v is a dead end. Blacken v.

$$Q = (u{:}\,3,\ y{:}\,3)$$

Step 7. Dequeue u**.** $\mathrm{Adj}[u] = [t, y]$. t is black — skip. y is gray — skip. Blacken u.

$$Q = (y{:}\,3)$$

Step 8. Dequeue y**.** $\mathrm{Adj}[y] = [u, x]$. Both black — skip both. Blacken y.

$$Q = (\,) \quad \text{— empty, so the while loop ends.}$$

The figures for this trace are **Figure 10.16** for steps 1 and 2, **Figure 10.17** for steps 3 and 4, **Figure 10.18** for steps 5 and 6 and **Figure 10.19** for steps 7 and 8, ending in **Figure 10.20**. In these drawings a white circle is a white vertex, a gray circle is gray, a filled black circle is black, the number inside each circle is d, and a thick edge is a tree edge — an edge $(\pi[v], v)$.

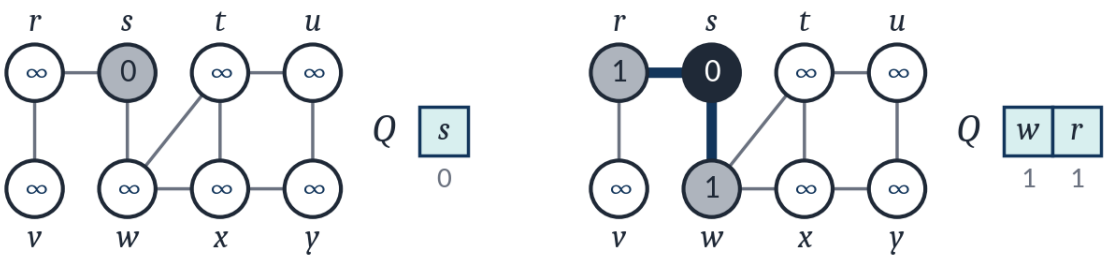


Figure 10.16 The BFS example, the first two states. **Left:** after initialization. Only s is gray and only s has a number, 0; every other vertex shows ∞. The queue holds s. **Right:** after step 1. Both neighbors of s are gray with distance 1, the two tree edges s–w and s–r are drawn thick, s is black, and the queue holds w then r — in that order, which fixes the order of $\mathrm{Adj}[s]$ as $[w, r]$.

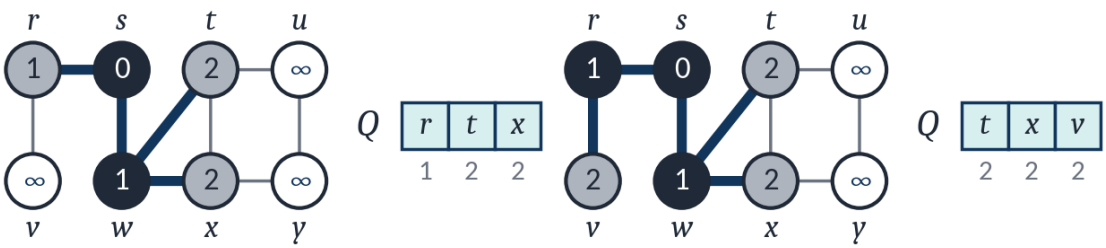


Figure 10.17 Steps 2 and 3. **Left:** w has been dequeued and blackened, discovering t and x at distance 2; the queue is r, t, x. **Right:** r has been dequeued and blackened, discovering v at distance 2; the queue is t, x, v — three vertices all at distance 2, the complete second ring.

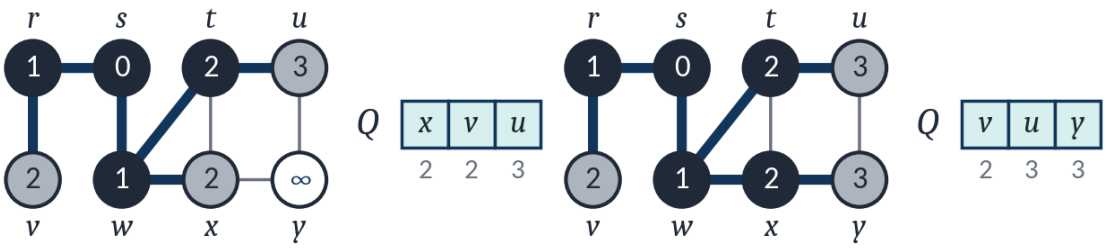


Figure 10.18 Steps 4 and 5. **Left:** t dequeued, discovering u at distance 3; the edge (t, x) was examined and rejected, because x was already gray with $d[x] = 2$. **Right:** x dequeued, discovering y at distance 3; the queue is v, u, y.

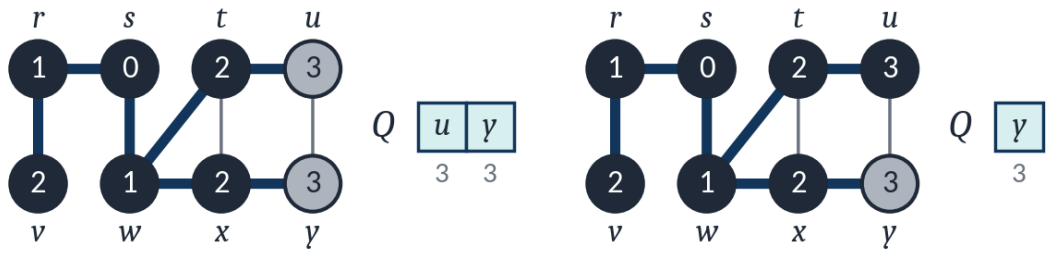


Figure 10.19 Steps 6 and 7. **Left:** v dequeued; its only neighbor r is black, so nothing is discovered and v simply turns black. **Right:** u dequeued; both its neighbors are already seen. The queue is down to y alone.

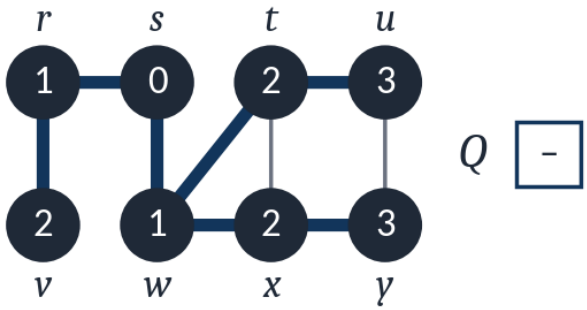


Figure 10.20 The result. Every vertex is black, the queue is empty, each vertex carries its distance from s, and the seven thick edges are the breadth-first tree. There are eight vertices and seven tree edges, exactly as **Figure 10.10**'s arithmetic requires of a tree.

10.7.5 Reading the answer off

Table 10.7 is the same trace as a table, which is the form to use whenever a trace is asked for — and Chapter 12 works problems of exactly this kind: one row per step, and the queue written out in full.

Table 10.7 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	$(\ )$

And **Table 10.8** is the answer: the two arrays the search was built to produce.

Table 10.8 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	$s\,w\,t$	$s\,w\,t\,u$	$s\,r\,v$	$s\,w$	$s\,w\,x$	$s\,w\,x\,y$

Read one of those paths out of π by hand, to be sure you can. Take $v = y$. $\pi[y] = x$; $\pi[x] = w$; $\pi[w] = s$; $\pi[s] =$ NIL, so stop. Writing them down as we went gives y, x, w, s; reversing gives s, w, x, y. Its length is 3 edges, and $d[y] = 3$. They agree, as they must.

10.7.6 The program

Listing 10.2 is the algorithm of **Figure 10.15** in C++, printing its own trace so that you can check it against **Table 10.7** line by line.

Listing 10.2 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 10.7** row by row.

```
// 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 10.2 (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 10.7 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 10.2 (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 10.7 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;
}
```

Its output, in full:

```
BFS from s
  init      enqueue s   Q = (s:0)
  step 1    dequeue s   Q = ( empty )
            discover w via s, d=1  Q = (w:1)
            discover r via s, d=1  Q = (w:1 r:1)
            s -> black
  step 2    dequeue w   Q = (r:1)
            skip s (already black)
            discover t via w, d=2  Q = (r:1 t:2)
            discover x via w, d=2  Q = (r:1 t:2 x:2)
            w -> black
  step 3    dequeue r   Q = (t:2 x:2)
            skip s (already black)
            discover v via r, d=2  Q = (t:2 x:2 v:2)
            r -> black
  step 4    dequeue t   Q = (x:2 v:2)
            discover u via t, d=3  Q = (x:2 v:2 u:3)
            skip w (already black)
            skip x (already gray )
            t -> black
  step 5    dequeue x   Q = (v:2 u:3)
            skip t (already black)
            skip w (already black)
            discover y via x, d=3  Q = (v:2 u:3 y:3)
            x -> black
  step 6    dequeue v   Q = (u:3 y:3)
            skip r (already black)
            v -> black
  step 7    dequeue u   Q = (y:3)
            skip t (already black)
            skip y (already gray )
            u -> black
  step 8    dequeue y   Q = ( empty )
            skip u (already black)
            skip x (already black)
            y -> black

result
  vertex :  r  s  t  u  v  w  x  y
  d      :  1  0  2  3  2  1  2  3
  pi     :  s  -  w  t  r  s  w  x

breadth-first tree edges (pi[v], v)
  (s,r) (w,t) (t,u) (r,v) (s,w) (w,x) (x,y)

shortest paths read out of pi
  s to r (1 edge): sr
  s to s (0 edges): s
  s to t (2 edges): swt
  s to u (3 edges): swtu
  s to v (2 edges): srv
  s to w (1 edge): sw
  s to x (2 edges): swx
  s to y (3 edges): swxy
```

Compare that output against **Table 10.7** and against **Figure 10.16** to **Figure 10.20**. The queue after each step is identical in all three: (w, r), then (r, t, x), then (t, x, v), then (x, v, u), then (v, u, y), then (u, y), then (y), then empty. The distances are identical. The seven tree edges are the seven thick edges in **Figure 10.20**. The program, the table and the drawings all agree, and that agreement is the point of doing all three.

10.7.7 Why it costs $O(V + E)$

The derivation takes five lines. Let us take them one at a time and put the reason under each.

1. Initialization takes $O(V)$. Lines 01–04 touch each vertex once and do three assignments, so $\Theta(V)$.

2. A vertex is enqueued only if it is white, and it stops being white at the same moment. Line 12 tests for white, and line 13 immediately paints it gray. Nothing ever paints a vertex white again. Therefore **each vertex is enqueued at most once**, and since each dequeue removes one vertex, each vertex is dequeued at most once too.

3. Enqueue and dequeue cost $O(1)$ each, so the total cost of all the queue operations is $O(V)$. There are at most $|V|$ of each, at constant cost apiece. (With `std::deque` or a plain array used circularly this is genuinely constant; with a `std::list` it is also constant. Using `std::vector::erase(begin())` as a dequeue would make it $\Theta(V)$ per operation and the whole search $\Theta(V^2)$ — one of the traps in §10.11.)

4. The adjacency list of a vertex is scanned when the vertex is dequeued, and only then. The inner loop, line 11, sits inside the body of the `while` loop, which runs once per dequeue. Since each vertex is dequeued at most once, $\mathrm{Adj}[u]$ is scanned at most once for each u.

5. The sum of the lengths of all the adjacency lists is $\Theta(E)$, so $O(E)$ time is spent scanning them. This is Consequence 2 of the handshake lemma Equation (10.1), §10.4.3: $\sum_v \deg(v) = 2|E|$ in an undirected graph, and $\sum_v \deg^+(v) = |E|$ in a directed one. Each entry examined costs $O(1)$ — a color test and, at most, four assignments and a push.

Adding the pieces:

$$T(n) = \underbrace{O(V)}_{\text{initialize}} + \underbrace{O(V)}_{\text{queue operations}} + \underbrace{O(E)}_{\text{scanning the lists}} = O(V+E). \tag{10.4}$$

The summary of Equation (10.4) is exact: *the total running time is $O(V+E)$ — linear in the size of the adjacency-list representation of G*. That final clause is the one to remember. The input **is** $\Theta(V+E)$ words long when stored as adjacency lists, so an algorithm that runs in $O(V+E)$ time is doing a constant amount of work per word of its input. You cannot do better than that, because you cannot read the input faster than that. Breadth-first search is *optimal*, not merely fast.

> **WARNING.** $O(V+E)$ is a claim about the **adjacency-list** representation. Run the identical algorithm over an adjacency **matrix** and line 11 becomes "for v from 1 to $|V|$", which costs $\Theta(V)$ per dequeued vertex whether the edges are there or not. The total becomes $\Theta(V^2)$. §10.9.8 measures exactly this, and on 8 000 vertices with 32 000 edges the matrix version takes about 136 times as long.

Why $O(V+E)$ and not $O(E)$? Because the two terms are independent. A graph may have many vertices and very few edges — ten thousand vertices and no edges at all is a legal graph — and the initialization must still touch every vertex, so the V term cannot be dropped. Conversely a graph may have few vertices and many edges — K_{1000} has a thousand vertices and half a million edges — and then the E term dominates. Neither term subsumes the other, so both are written. When the graph is connected we know $|E| \geq |V| - 1$, and then $O(V+E) = O(E)$; many papers assume connectivity and write $O(E)$ for that reason.

10.7.8 The breadth-first tree

The π array is not just a pile of pointers. It is a tree, called the **predecessor subgraph**.

> **DEFINITION.** After running BFS from s, the **predecessor subgraph** of G is $G_\pi = (V_\pi, E_\pi)$ where

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}, \qquad E_\pi = \big\{(\pi[v], v) \in E : v \in V_\pi - \{s\}\big\}. \tag{10.5}$$

> G_π is called the **breadth-first tree**, and its edges are called **tree edges**.

Read the two set-builders of Equation (10.5) out loud. V_π is "the vertices that have a predecessor, plus the source" — and since a vertex gets a predecessor exactly when it is discovered, V_π is precisely the set of vertices the search reached. E_π is "for each such vertex other than the source, the one edge that discovered it". So $|E_\pi| = |V_\pi| - 1$: every vertex in the tree except the root contributes exactly one edge.

The breadth-first tree has three properties, and we can now see each of them in our worked example.

- **V_π consists of the vertices reachable from s.** In our example all eight vertices were reached, so $V_\pi = V$.
- **$|E_\pi| = |V_\pi| - 1$, and G_π is connected, so G_π is a tree.** Our example produced seven tree edges for eight vertices — read them off the program output: (s, r), (s, w), (w, t), (w, x), (r, v), (t, u), (x, y). Seven is eight minus one. It is connected because every vertex in it has a path to s along predecessors. By §10.4.11, connected plus $|E| = |V| - 1$ makes it a tree.
- **For every $v \in V_\pi$ there is a unique simple path from s to v in G_π, and it is also a shortest path from s to v in G.** Uniqueness is a property of trees. That it is *shortest* is the theorem of the next section.

The breadth-first tree is a **spanning tree** of the component containing s — a spanning subgraph (§10.4.10) that happens to be a tree. This is the first of several algorithms in the book that produce spanning trees; Chapter 11 produces the *cheapest* one.

Compare the tree edges with the graph's nine edges to see what BFS threw away. The edges of G are (r, s), (r, v), (s, w), (t, u), (t, w), (t, x), (u, y), (w, x), (x, y). Seven became tree edges. The two left over are (t, x) — rejected at step 4, because x was already gray — and (u, y) — rejected at step 7, for the same reason. In general, $|E| - (|V_\pi| - 1)$ edges are rejected, here $9 - 7 = 2$. Every rejected edge joins two vertices whose distances differ by at most one; that is Corollary 10.5 below, and it is the fact bipartiteness testing rests on.

10.7.9 BFS computes shortest distances: the proof

We come to the reason BFS matters. Stated as a list of properties, the claim is:

Given a graph $G = (V, E)$, BFS discovers all vertices reachable from a source vertex s. It computes the shortest distance to all reachable vertices. It computes a breadth-first tree that contains all such reachable vertices. For any vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a shortest path in G.

Those four sentences are a theorem, and we shall prove it. Recall from §10.4.5 that $\delta(s, v)$ denotes the true shortest-path distance — the smallest number of edges on any path from s to v, or ∞ if none exists. The claim is that when BFS finishes, $d[v] = \delta(s, v)$ for every v. The proof is in four pieces: a fact about the graph, an upper-bound-free half, a fact about the queue, and then the theorem.

Lemma 10.1 (a fact about the graph, no algorithm involved). *Let $G = (V, E)$ be a graph and $s \in V$. For every edge $(u, v) \in E$,*

$$\delta(s, v) \leq \delta(s, u) + 1. \tag{10.6}$$

Proof. Two cases. If u is reachable from s, take a shortest path from s to u — it has $\delta(s,u)$ edges — and append the edge (u,v). The result is a path from s to v with $\delta(s,u)+1$ edges. The shortest path to v is no longer than that particular path, so $\delta(s,v) \le \delta(s,u)+1$. If u is not reachable from s then $\delta(s,u)=\infty$ and the inequality holds trivially, since nothing exceeds ∞. ■

In words: *taking one more step cannot get you more than one step further from home.* Obvious, and indispensable.

Lemma 10.2 (BFS never underestimates). *At the end of BFS from s, we have $d[v] \ge \delta(s,v)$ for every $v \in V$.*

Proof. By induction on the number of vertices that have been enqueued so far — that is, we show the statement "$d[v] \ge \delta(s,v)$ for all v" is true after the initialization and is preserved by every enqueue.

Base. After initialization, $d[s]=0=\delta(s,s)$, and $d[v]=\infty \ge \delta(s,v)$ for every other v. True.

Step. Consider the moment a white vertex v is discovered while scanning $\mathrm{Adj}[u]$, so that line 14 sets $d[v] \leftarrow d[u]+1$. The inductive hypothesis gives $d[u] \ge \delta(s,u)$. Therefore

$$d[v] = d[u]+1 \ge \delta(s,u)+1 \ge \delta(s,v),$$

where the last inequality is Lemma 10.1, Equation (10.6), applied to the edge (u,v). And no other line of the algorithm ever changes any d value: once set, $d[v]$ is never touched again, because v is no longer white and line 12 guards the only assignment. So the property is preserved. ■

So BFS's answer is never too small. Half the theorem. The other half needs a fact about the *queue*, and this is the heart of the matter.

Lemma 10.3 (the queue holds at most two distinct distances, in order). *Suppose that during the execution of BFS the queue contains the vertices $\langle v_1, v_2, \dots, v_r\rangle$, with v_1 at the head. Then*

$$d[v_1] \le d[v_2] \le \cdots \le d[v_r] \le d[v_1]+1. \tag{10.7}$$

Proof. By induction on the number of queue operations performed.

Base. Immediately after initialization the queue is $\langle s\rangle$: a single element, so both chains of inequalities read $d[s] \le d[s] \le d[s]+1$. True.

Step, for a dequeue. Suppose the property holds for $\langle v_1, \dots, v_r\rangle$ and we remove the head v_1, leaving $\langle v_2, \dots, v_r\rangle$ (if $r=1$ the queue becomes empty and there is nothing to prove). We must check the new head, v_2. We need $d[v_r] \le d[v_2]+1$. We know $d[v_r] \le d[v_1]+1$ and $d[v_1] \le d[v_2]$, so $d[v_r] \le d[v_2]+1$. The internal inequalities $d[v_2] \le \cdots \le d[v_r]$ are inherited unchanged. True.

Step, for an enqueue. Suppose we are scanning $\mathrm{Adj}[u]$ for the vertex u that was most recently dequeued, and we discover a white v, setting $d[v]=d[u]+1$ and appending v to the queue, making it $\langle v_1, \dots, v_r, v\rangle$. Two things must be checked.

First, $d[v_r] \le d[v]$. At the moment u was dequeued it was the head of the queue, so by the inductive hypothesis at that moment every vertex then in the queue had d value at least $d[u]$ and at most $d[u]+1$. Every vertex enqueued since then was given the value $d[u]+1$. So every vertex currently in the queue has d value in $\{d[u], d[u]+1\}$; in particular $d[v_r] \le d[u]+1 = d[v]$. Good.

Second, $d[v] \leq d[v_1] + 1$. From the same observation, $d[v_1] \geq d[u]$, so $d[v_1] + 1 \geq d[u] + 1 = d[v]$. Good. ■

The lemma's inequality Equation (10.7) says something very concrete, and you can see it in every row of **Table 10.7**: **the queue is always sorted by distance, and it always holds at most two different distances, k and $k + 1$.** Look at step 3: the queue is $(t: 2, x: 2, v: 2)$ — one distance. Step 4: $(x: 2, v: 2, u: 3)$ — two adjacent distances, in order. Never $(u: 3, v: 2)$, and never 2,4 with no 3. That is the discipline that makes the rings real.

Corollary. *Vertices are dequeued in non-decreasing order of d, and each vertex is enqueued at most once, so if v is enqueued before w then $d[v] \leq d[w]$.*

Theorem 10.4 (correctness of breadth-first search). *Let BFS be run on a graph $G = (V, E)$ from a source s. Then BFS discovers every vertex $v \in V$ that is reachable from s, it discovers no vertex that is unreachable, and at termination $d[v] = \delta(s, v)$ for every $v \in V$. Moreover, for every reachable $v \neq s$, one shortest path from s to v is the path from s to $\pi[v]$ followed by the edge $(\pi[v], v)$.*

Proof. Suppose, for contradiction, that some vertex receives a d value different from its true distance. By Lemma 10.2 we know $d[v] \geq \delta(s, v)$ always, so any wrong value must be **too large**. Among all vertices with $d[v] \neq \delta(s, v)$, let v be one with the smallest $\delta(s, v)$; if several tie, pick any. Then

$$d[v] > \delta(s, v). \tag{10.8}$$

Note first that $v \neq s$, because $d[s] = 0 = \delta(s, s)$. Note second that v is reachable from s: if it were not, $\delta(s, v) = \infty$, and since $d[v] \geq \delta(s, v)$ we would have $d[v] = \infty$ too and no discrepancy. So $\delta(s, v)$ is a finite number at least 1.

Because v is reachable and $v \neq s$, some shortest path from s to v exists; let u be the vertex just before v on it. Then $(u, v) \in E$ and

$$\delta(s, v) = \delta(s, u) + 1.$$

Since $\delta(s, u) = \delta(s, v) - 1 < \delta(s, v)$, and v was chosen with the *smallest* wrong distance, u cannot be wrong. So $d[u] = \delta(s, u)$. Substituting into Equation (10.8):

$$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1. \tag{10.9}$$

Now consider the moment the algorithm dequeues u — which it does, because $d[u]$ is finite, so u was discovered, so u was enqueued, and everything enqueued is eventually dequeued. At that moment the algorithm scans $\text{Adj}[u]$ and examines the edge (u, v). There are three possibilities for v's color at that instant, and we shall rule out all three.

Case 1: v is white. Then line 14 executes and sets $d[v] \leftarrow d[u] + 1$, contradicting Equation (10.9), which says $d[v] > d[u] + 1$.

Case 2: v is black. A black vertex has already been dequeued. By the corollary to Lemma 10.3, vertices are dequeued in non-decreasing order of d, so v being dequeued before u gives $d[v] \leq d[u]$, which contradicts Equation (10.9) even more strongly.

Case 3: v is gray. Then v was discovered while scanning the adjacency list of some vertex w that was dequeued before u (or is u itself, but then Case 1 applied at that earlier moment). Since w

was dequeued no later than u, the corollary gives $d[w] \leq d[u]$. And v's value was set from w: $d[v] = d[w] + 1 \leq d[u] + 1$. Again this contradicts Equation (10.9).

All three cases are impossible, so no such v exists, and $d[v] = \delta(s, v)$ for every vertex.

It remains to note the two claims about *which* vertices are discovered. If v is reachable then $\delta(s, v)$ is finite, so $d[v]$ is finite, so $d[v]$ was assigned by line 14, which happens only on discovery — hence v is discovered. If v is unreachable then $\delta(s, v) = \infty$, so $d[v] = \infty$, so line 14 never fired for v and it was never discovered.

Finally, the path claim. For a reachable $v \neq s$ we have $d[v] = d[\pi[v]] + 1$ by construction (line 14 set it so), and we have just shown d equals δ, so $\delta(s, v) = \delta(s, \pi[v]) + 1$. So a shortest path to $\pi[v]$ followed by the edge $(\pi[v], v)$ is a path to v of length $\delta(s, v)$ — a shortest path. By induction on $d[v]$, following π all the way back to s yields a shortest path from s to v; and that is exactly the unique tree path in G_π, since G_π is a tree whose edges are the pairs $(\pi[v], v)$. ■

> **KEY IDEA, in plain language.** Why does a queue find shortest paths? Because a queue serves the frontier in the order it was discovered, and the order of discovery *is* the order of distance. The algorithm therefore finishes every vertex at distance k before touching any vertex at distance $k + 1$, so when it first touches a vertex it is standing on the closest possible approach. A stack would break this — it would plunge to distance 17 before finishing distance 1 — and that is precisely why depth-first search does **not** compute shortest paths.

Corollary 10.5 (every edge joins levels that differ by at most one). *For every edge (u, v) of an undirected graph, $|d[u] - d[v]| \leq 1$ after BFS from any source that reaches both.*

Proof. By Theorem 10.4, $d = \delta$. By Lemma 10.1, Equation (10.6), applied to (u, v), $\delta(s, v) \leq \delta(s, u) + 1$; applied to (v, u), $\delta(s, u) \leq \delta(s, v) + 1$. Together these say the two values differ by at most one. ■

Corollary 10.5 classifies every edge of an undirected graph after BFS into exactly two kinds: a **tree edge**, joining level k to level $k + 1$ and used to discover the vertex at level $k + 1$; or a **cross edge**, joining two vertices at the same level, or joining levels k and $k + 1$ without having been the discovering edge. What is *impossible* is an edge joining level k to level $k + 2$ — if such an edge existed, the far vertex would have been discovered a ring earlier. This is the fact §10.9.3 uses: a same-level edge is exactly an odd cycle, and hence exactly an obstruction to bipartiteness.

10.8 Depth-First Search

10.8.1 The labyrinth, the string, and the can of paint

The classic metaphor for depth-first search [2] extends the one for BFS by exactly one object, and the extra object is the whole difference:

A depth-first search (DFS) in an undirected graph G is like wandering in a labyrinth with a string **and a can of paint**. We start at vertex s, tying the end of our string to the point and painting s "visited (discovered)". Next we label s as our current vertex called u. Now, we travel along an arbitrary edge (u, v). If edge (u, v) leads us to an already visited vertex v we return to u. If vertex v is unvisited, we unroll our string, move to v, paint v "visited", set v as our current vertex, and repeat the previous steps.

Eventually, we will get to a point where all incident edges on u lead to visited vertices. We then backtrack by unrolling our string to a previously visited vertex v. Then v becomes our current vertex and we repeat the previous steps. Then, if all incident edges on v lead to visited vertices,

we backtrack as we did before. We continue to backtrack along the path we have traveled, finding and exploring unexplored edges, and repeating the procedure.

In breadth-first search you never moved: you stood at s and let the string out in circles. In depth-first search **you walk**. You pick a corridor and follow it as far as it goes, then the next corridor from there, and so on, until you hit a room all of whose exits lead somewhere you have already painted. Only then do you retreat — one room at a time, trying each unexplored exit as you pass it.

The string is still there, and it is now doing a second job. In BFS the string measured distance. In DFS the string **is the route back**: it records the exact sequence of rooms you came through, and retreating means following it backward. That sequence, held in order with the most recent room first, is a **stack** — and in the recursive form of the algorithm it is the call stack of the program itself, which is why the recursive form is so short.

The paint is the color of §10.6, and DFS uses all three shades where BFS arguably needed only two. Precisely:

- *A vertex is **white** if it is undiscovered.*
- *A vertex is **gray** if it has been discovered but not all of its edges have been discovered.* Equivalently: you are currently standing in that room, or you are somewhere deeper down a corridor that started there. The gray vertices are exactly the rooms your string passes through.
- *A vertex is **black** after all of its adjacent vertices have been discovered (the adjacency list was examined completely).* You have finished with that room and pulled your string back past it.

> **KEY IDEA.** The gray vertices in a DFS form a **path**, not a ring: exactly the path from the root of the current tree to the vertex you are standing on. In a phrase: *the gray vertices form a linear chain, which corresponds to a stack of vertices that have not been exhaustively explored.* This single observation is the origin of every DFS theorem in this chapter. "Is v gray?" means "is v an ancestor of where I am standing?", and that is why a gray vertex ahead of you means a cycle.

10.8.2 The algorithm

Unlike BFS, DFS is naturally written as two procedures, because it does two things: it *starts* searches, and it *conducts* them. **Figure 10.21** gives both.

```
DFS (G)
for each vertex u ∈ V[G]
      do color[u] ← WHITE
time ← 0
for each vertex u ∈ V[G]
      do if color[u] = WHITE
            then DFS-VISIT (u)

DFS-VISIT (u)
color[u] ← GRAY                    ▷ White vertex u discovered.
d[u] ← time                        ▷ Mark with discovery time.
time ← time + 1                    ▷ Tick global time.
for each v ∈ Adj[u]                ▷ Explore all edges (u, v).
      do if color[v] = WHITE
            then DFS-VISIT (v)
color[u] ← BLACK                   ▷ Blacken u; it is finished.
f[u] ← time                        ▷ Mark with finishing time.
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.

The structure is this:

- *Initialize — color all vertices white.*
- *Visit each and every white vertex using DFS-Visit.*
- *Each call to DFS-Visit(u) roots a new tree of the depth-first forest at vertex u.*

That second bullet is the great structural difference from BFS. **Breadth-first search takes a source and explores one component. Depth-first search takes no source and explores the whole graph.** The outer loop of DFS guarantees that if the graph is disconnected, or directed in a way that leaves some vertices unreachable from the first root, those vertices still get searched — from a new root. The result is therefore not a tree but a **forest**, with one tree per root.

> **WARNING — the clock is off by one between the procedure and the worked example.** In **Figure 10.21** the clock is started at $time \leftarrow 0$ and each stamp is taken *before* the increment ($d[u] \leftarrow time$, then $time \leftarrow time + 1$), so the first vertex discovered would receive $d = 0$. The worked example, **Figure 10.22** onwards, starts at 1. Both conventions are perfectly correct and neither changes any theorem — every statement in this chapter is about the *order* of the timestamps, never their absolute values. To match the figures, and CLRS [1], this book increments first: $time \leftarrow time + 1$, then $d[u] \leftarrow time$. Whenever you report timestamps, say which convention you are using; a 1-based answer and a 0-based answer are both right.

In the notation of this book, with the clock incremented first, the two procedures are **Pseudocode 10.2**:

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)
for each vertex u in G.V
    color[u] = WHITE
    pi[u] = NIL
time = 0
for each vertex u in G.V
    if color[u] == WHITE
        DFS-VISIT(u)

DFS-VISIT(u)
color[u] = GRAY                 // white vertex u has just been discovered
time = time + 1
d[u] = time                     // stamp the discovery time
for each v in Adj[u]            // explore every edge (u, v)
    if color[v] == WHITE
        pi[v] = u
        DFS-VISIT(v)
color[u] = BLACK                // u is finished
time = time + 1
f[u] = time                     // stamp the finishing time
```

Notice how much smaller this is than BFS. There is no queue, because the recursion *is* the stack; there is no distance array, because DFS does not compute distances. What it computes instead is two timestamps per vertex, and everything in §10.8.7 onward is squeezed out of those.

10.8.3 The two timestamps

> **DEFINITION.** DFS maintains a single global clock, a counter that only ever increases. Each vertex u receives two stamps from it: the **discovery time** $d[u]$, the reading of the clock at the instant u turned gray, and the **finishing time** $f[u]$, the reading at the instant u turned black. For every vertex, $d[u] < f[u]$.

The invariant — *for every vertex u, the inequality $d[u] < f[u]$ must hold* — has an immediate reason: the clock advances between the two stamps, at the very least by the one tick spent finishing u, and never runs backward.

Two numerical facts follow, and they are used constantly.

The clock ends at exactly $2|V|$. Each vertex is discovered exactly once and finished exactly once, and each event advances the clock by one. So there are $2|V|$ ticks, and the timestamps are exactly the integers 1 through $2|V|$, each used once. In our six-vertex example the stamps will be 1 through 12.

A vertex is white before $d[u]$, gray between $d[u]$ and $f[u]$, and black after $f[u]$. This restatement makes the timestamps into *intervals*: vertex u "occupies" the closed interval $[d[u], f[u]]$ on the clock, and it is gray for exactly that interval. The structure of those intervals is the parenthesis theorem of §10.8.8.

10.8.4 A worked example, one step at a time

The example is a **directed** graph — DFS is where directedness begins to matter — on six vertices:

$$V = \{u, v, w, x, y, z\}, \qquad E = \{u \to v,\ u \to x,\ v \to y,\ w \to y,\ w \to z,\ x \to v,\ y \to x,\ z \to z\}.$$

Eight arcs, including one self-loop $z \to z$. The in- and out-degrees are those of **Table 10.1**. The outer loop considers the vertices in the order u, v, w, x, y, z, and each adjacency list is in the order written above.

Each numbered step below is one clock tick or one edge examination. This is the level of detail at which DFS becomes obvious; anything coarser and it remains mysterious.

Step 1. The outer loop reaches u, which is white. Call DFS-Visit(u). Paint u gray; $d[u] = 1$. **Step 2.** Scan Adj$[u] = [v, x]$. First v: white. So (u, v) is a **tree edge**; set $\pi[v] = u$ and recurse. Paint v gray; $d[v] = 2$. **Step 3.** Scan Adj$[v] = [y]$. y is white. (v, y) is a **tree edge**; $\pi[y] = v$; recurse. Paint y gray; $d[y] = 3$. **Step 4.** Scan Adj$[y] = [x]$. x is white. (y, x) is a **tree edge**; $\pi[x] = y$; recurse. Paint x gray; $d[x] = 4$. **Step 5.** Scan Adj$[x] = [v]$. v is **gray**. So v is an ancestor of x — indeed $v \to y \to x$ is the chain of gray vertices right now — and (x, v) is a **back edge**. Do not recurse. This is the algorithm detecting the directed cycle $v \to y \to x \to v$. **Step 6.** Adj$[x]$ is exhausted. Paint x black; $f[x] = 5$. Return to y. **Step 7.** Adj$[y]$ is exhausted. Paint y black; $f[y] = 6$. Return to v. **Step 8.** Adj$[v]$ is exhausted. Paint v black; $f[v] = 7$. Return to u. **Step 9.** Back in Adj$[u]$, the next neighbor is x. It is **black**, and $d[u] = 1 < 4 = d[x]$, so x is a *descendant* of u and (u, x) is a **forward edge**. Do not recurse. **Step 10.** Adj$[u]$ is exhausted. Paint u black; $f[u] = 8$. Return to the outer loop. The first tree of the forest is complete: it contains u, v, y, x. **Step 11.** The outer loop tries v — black, skip. Then w — white. Call DFS-Visit(w), rooting a **second tree**. Paint w gray; $d[w] = 9$. **Step 12.** Scan Adj$[w] = [y, z]$. First y: **black**, and $d[w] = 9 > 3 = d[y]$, so y is neither ancestor nor descendant of w — it is in a different tree entirely. (w, y) is a **cross edge**. **Step 13.** Next z: white. (w, z) is a **tree edge**; $\pi[z] = w$; recurse. Paint z gray; $d[z] = 10$. **Step 14.** Scan Adj$[z] = [z]$. z is **gray** — it is itself, and it is currently gray. So (z, z) is a **back edge**, and the self-loop is correctly reported as a directed cycle of length 1. No special case was needed. **Step 15.** Adj$[z]$ exhausted. Paint z black; $f[z] = 11$. **Step 16.** Adj$[w]$ exhausted. Paint w black; $f[w] = 12$. The outer loop finds x, y, z all black, and stops. The clock ended at $12 = 2 \times 6$, as predicted.

The figures for these sixteen steps are **Figure 10.22**, **Figure 10.23** and **Figure 10.24** — six panels, six panels, and four panels, one per step, in reading order.

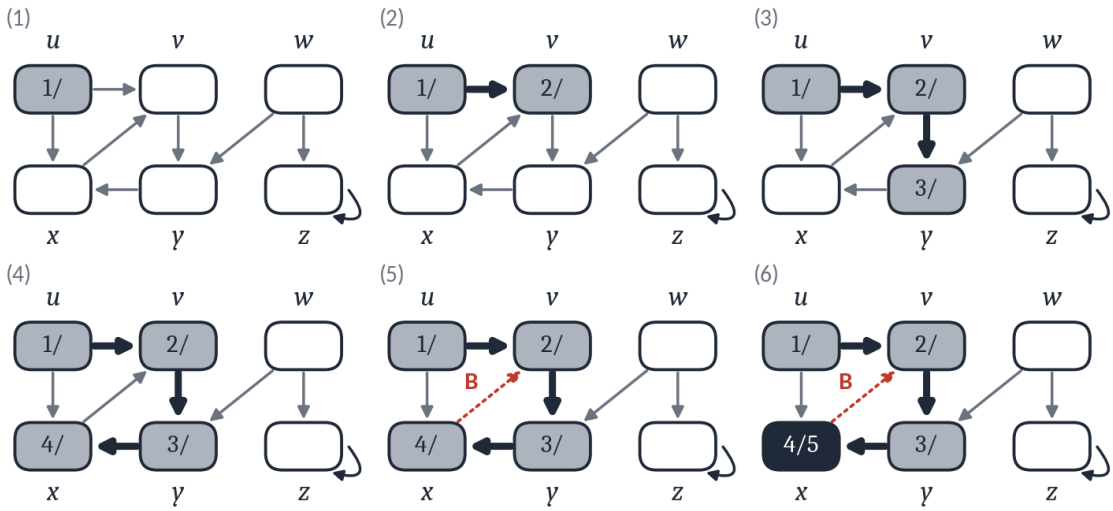


Figure 10.22 Steps 1 to 6 of the DFS example, in reading order: u turns gray with $d = 1$; v is discovered as a tree edge with $d = 2$; y with $d = 3$; x with $d = 4$; the edge (x, v) is examined, found to point at a gray vertex, and labeled **B** for back edge, drawn as a dotted arrow; and finally x turns black with $f = 5$, shown as 4/5.

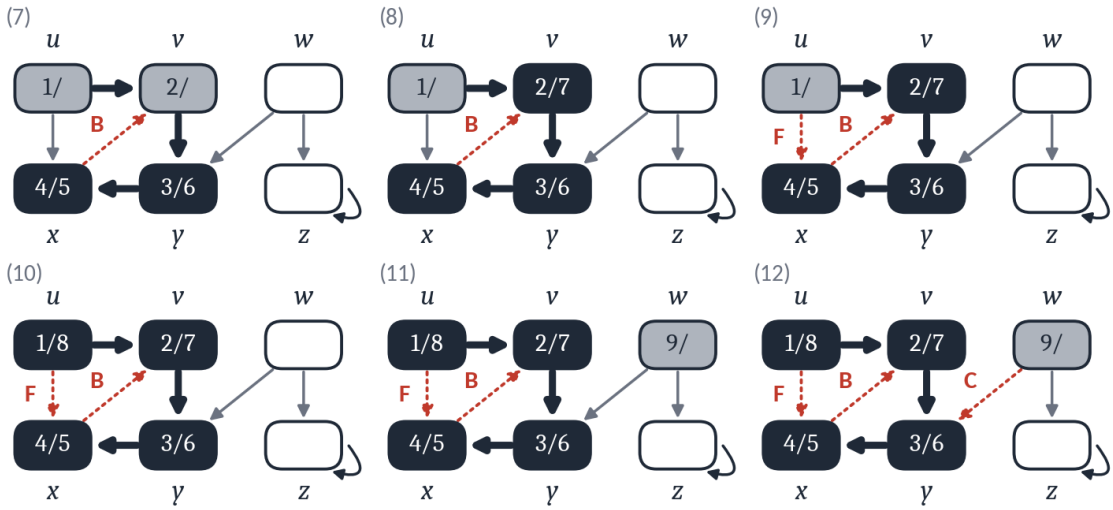


Figure 10.23 Steps 7 to 12. The recursion unwinds, blackening y at $3/6$ and v at $2/7$; back in u's adjacency list the edge (u, x) is examined and labeled **F** for forward edge; u finishes at $1/8$, completing the first tree; the outer loop then roots a second tree at w with $d = 9$; and the edge (w, y) is labeled **C** for cross edge, since y belongs to the first tree.

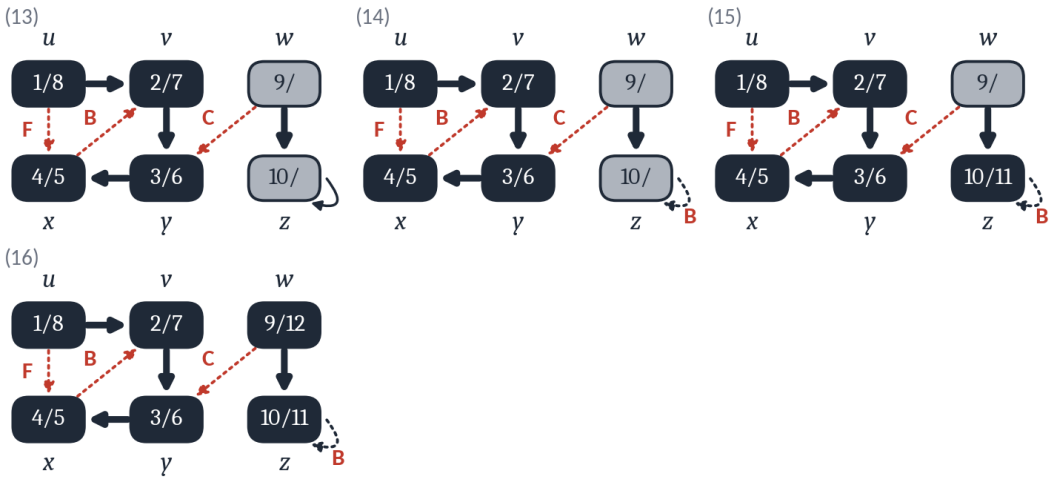


Figure 10.24 Steps 13 to 16. z is discovered from w as a tree edge with $d = 10$; the self-loop (z, z) is examined and labeled **B**, because at that instant z is gray; z finishes at $10/11$; and w finishes at $9/12$. Every vertex is now black and the clock has reached $12 = 2|V|$.

Table 10.9 is the answer, and **Table 10.10** is the edge classification. Both are worth copying out by hand once.

Table 10.9 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 10.10 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

10.8.5 The depth-first forest

The predecessor subgraph for DFS is defined slightly differently from that of BFS:

DEFINITION. After a DFS, the predecessor subgraph is $G_\pi = (V, E_\pi)$ where

$$E_\pi = \{(\pi[v], v) \in E : v \in V \text{ and } \pi[v] \neq \text{NIL}\}.$$

(10.10)

G_π is the **depth-first forest**; its connected pieces are the **depth-first trees**, and its edges are the **tree edges**.

The difference between Equation (10.10) and BFS's Equation (10.5) is that the vertex set here is *all* of V, not just the reachable part. Because the outer loop of DFS visits every vertex, no vertex is left out; a vertex reachable from nothing simply becomes the root of its own one-vertex tree. So the depth-first forest is always a **spanning** forest of G (§10.4.10).

In our example $E_\pi = \{(u, v), (v, y), (y, x), (w, z)\}$ — four tree edges, six vertices, two trees. The arithmetic is $|V| - (\text{number of trees}) = 6 - 2 = 4$, which is the general rule: a forest with k trees on n vertices has $n - k$ edges.

The two trees are

$$\underbrace{u \to v \to y \to x}_{\text{tree rooted at } u}, \qquad \underbrace{w \to z}_{\text{tree rooted at } w}.$$

10.8.6 Without recursion

Recursion is elegant but it uses the program's call stack, which on most systems is a few megabytes — enough for a few tens of thousands of nested calls, no more. A depth-first search on a graph that happens to contain a long path will overflow it. A million-vertex path graph will crash

any recursive DFS. So it is worth knowing the iterative form, and writing it is instructive, because it exposes the one thing the recursion was hiding.

The thing the recursion hides is this: when you come back to a vertex after a recursive call, you must resume its adjacency list **where you left off**, not from the beginning. A stack of vertices alone is not enough; you need a stack of *positions*. **Listing 10.3** keeps, for each gray vertex, both the vertex and an index into its adjacency list.

Listing 10.3 Depth-first search with an explicit stack, printing the stack at every step. The `Frame` struct carries the resume position `next` — without it, returning to a vertex would restart its adjacency list and the search would loop.

```
// Depth-first search without recursion: an explicit stack, the stack printed
// at every step. Same graph and same timestamps as the recursive version.
//   V = {u,v,w,x,y,z}, E = {u->v, u->x, v->y, w->y, w->z, x->v, y->x, z->z}
#include <iostream>
#include <vector>
#include <string>
using namespace std;

enum Color { WHITE, GRAY, BLACK };
static const string NAME = "uvwxyz";

struct Frame { int u; size_t next; };      // vertex, and how far into Adj[u] we got

static string stackToString(const vector<Frame>& S) {
    if (S.empty()) return "[ ]";
    string s = "[";
    for (size_t i = 0; i < S.size(); ++i) {
        if (i) s += " ";
        s += NAME[S[i].u];
    }
    return s + "]";
}

int main() {
    const int n = 6;
    vector<vector<int>> adj(n);
    adj[0] = {1,3};    // u -> v, x
    adj[1] = {4};      // v -> y
    adj[2] = {4,5};    // w -> y, z
    adj[3] = {1};      // x -> v
    adj[4] = {3};      // y -> x
    adj[5] = {5};      // z -> z

    vector<Color> color(n, WHITE);
    vector<int> d(n,0), f(n,0), pi(n,-1);
    int time = 0, step = 0;

    cout << "DFS, iterative (explicit stack)\n";
    for (int root = 0; root < n; ++root) {
        if (color[root] != WHITE) continue;
        vector<Frame> S;
        color[root] = GRAY; d[root] = ++time;
        S.push_back({root, 0});
        cout << "  step " << ++step << "  push " << NAME[root]
             << " (root), d=" << d[root] << "   stack = " << stackToString(S) << "\n";
```

Listing 10.3 (Continued) *Depth-first search with an explicit stack, printing the stack at every step. The Frame struct carries the resume position next — without it, returning to a vertex would restart its adjacency list and the search would loop.*

```
        while (!S.empty()) {
            Frame& top = S.back();
            if (top.next < adj[top.u].size()) {
                int v = adj[top.u][top.next++];        // take the next unexplored edge
                const char* kind = color[v] == WHITE ? "tree"
                                 : color[v] == GRAY  ? "back"
                                 : d[top.u] < d[v]    ? "forward" : "cross";
                cout << "  step " << ++step << "  edge (" << NAME[top.u] << ","
                     << NAME[v] << ") " << kind;
                if (color[v] == WHITE) {
                    color[v] = GRAY; d[v] = ++time; pi[v] = top.u;
                    S.push_back({v, 0});
                    cout << ": push " << NAME[v] << ", d=" << d[v];
                }
                cout << "   stack = " << stackToString(S) << "\n";
            } else {
                int u = top.u;
                S.pop_back();                          // Adj[u] exhausted: u finishes
                color[u] = BLACK; f[u] = ++time;
                cout << "  step " << ++step << "  pop  " << NAME[u]
                     << ", f=" << f[u] << "        stack = " << stackToString(S) << "\n";
            }
        }
    }

    cout << "\ntimestamps\n  vertex :";
    for (int i = 0; i < n; ++i) cout << "   " << NAME[i];
    cout << "\n  d/f    :";
    for (int i = 0; i < n; ++i) cout << " " << d[i] << "/" << f[i];
    cout << "\n\ntree edges\n  ";
    for (int v = 0; v < n; ++v)
        if (pi[v] >= 0) cout << "(" << NAME[pi[v]] << "," << NAME[v] << ") ";
    cout << "\n";
    return 0;
}
```

Its output is the trace of §10.8.4 with the stack made visible:

```
DFS, iterative (explicit stack)
  step 1  push u (root), d=1   stack = [u]
  step 2  edge (u,v) tree: push v, d=2   stack = [u v]
  step 3  edge (v,y) tree: push y, d=3   stack = [u v y]
  step 4  edge (y,x) tree: push x, d=4   stack = [u v y x]
  step 5  edge (x,v) back   stack = [u v y x]
  step 6  pop  x, f=5        stack = [u v y]
  step 7  pop  y, f=6        stack = [u v]
  step 8  pop  v, f=7        stack = [u]
  step 9  edge (u,x) forward   stack = [u]
  step 10  pop  u, f=8        stack = [ ]
  step 11  push w (root), d=9   stack = [w]
  step 12  edge (w,y) cross   stack = [w]
  step 13  edge (w,z) tree: push z, d=10   stack = [w z]
  step 14  edge (z,z) back   stack = [w z]
  step 15  pop  z, f=11        stack = [w]
  step 16  pop  w, f=12        stack = [ ]

timestamps
  vertex :   u   v   w   x   y   z
  d/f    : 1/8 2/7 9/12 4/5 3/6 10/11

tree edges
  (u,v) (y,x) (v,y) (w,z)
```

The sixteen steps are the sixteen steps of §10.8.4 and the sixteen panels of **Figure 10.22** to **Figure 10.24**. The timestamps are identical to **Table 10.9**. And now compare the stack column with the queue column of **Table 10.7**: at step 4 the DFS stack is $[u\ v\ y\ x]$ — four vertices deep, a *chain*, exactly the "linear chain of gray vertices" of §10.8.1. A BFS queue never looks like that.

And here, for completeness, is the recursive form, which is shorter and is what you should write unless the graph is deep.

Listing 10.4 Depth-first search, recursive, on the example of §10.8.4. It prints the trace, the timestamps, the parenthesization of §10.8.8, the forest, and the classification of every arc.

```
// Depth-first search, recursive form, on the directed graph of the
// DFS example of Section 10.8.4:  V = {u,v,w,x,y,z}
//   E = {u->v, u->x, v->y, w->y, w->z, x->v, y->x, z->z}
// Prints discovery/finishing times, the parenthesization, the depth-first
// forest, and the classification of every edge.
#include <iostream>
#include <vector>
#include <string>
using namespace std;

enum Color { WHITE, GRAY, BLACK };
static const string NAME = "uvwxyz";

struct Digraph {
    int n;
    vector<vector<int>> adj;
    explicit Digraph(int n_) : n(n_), adj(n_) {}
    void addArc(int u, int v) { adj[u].push_back(v); }
};

struct DFS {
    const Digraph& G;
    vector<Color> color;
    vector<int> d, f, pi;
    int time = 0;
    string parens;                      // the parenthesis structure
    vector<string> classified;          // one line per edge
    vector<int> finishOrder;            // vertices in order of finishing

    explicit DFS(const Digraph& g)
        : G(g), color(g.n, WHITE), d(g.n, 0), f(g.n, 0), pi(g.n, -1) {}

    void run() {
        for (int u = 0; u < G.n; ++u)
            if (color[u] == WHITE) visit(u);
    }

    void visit(int u) {
        color[u] = GRAY;
        d[u] = ++time;                                  // discovery time
        parens += "(" + string(1, NAME[u]) + " ";
        cout << "  d[" << NAME[u] << "] = " << d[u] << "   " << NAME[u]
             << " -> gray\n";
```

Listing 10.4 (Continued) *Depth-first search, recursive, on the example of §10.8.4. It prints the trace, the timestamps, the parenthesization of §10.8.8, the forest, and the classification of every arc.*

```
        for (int v : G.adj[u]) {
            string kind;
            if (color[v] == WHITE)      kind = "tree";
            else if (color[v] == GRAY)  kind = "back";     // v is an ancestor
            else if (d[u] < d[v])        kind = "forward";  // v is a descendant
            else                         kind = "cross";
            classified.push_back("(" + string(1,NAME[u]) + "," + string(1,NAME[v])
                                 + ")  " + kind);
            cout << "    edge (" << NAME[u] << "," << NAME[v] << ") is a "
                 << kind << " edge\n";
            if (color[v] == WHITE) { pi[v] = u; visit(v); }
        }

        color[u] = BLACK;
        f[u] = ++time;                                  // finishing time
        finishOrder.push_back(u);
        parens += string(1, NAME[u]) + ") ";
        cout << "  f[" << NAME[u] << "] = " << f[u] << "   " << NAME[u]
             << " -> black\n";
    }
};

int main() {
    Digraph G(6);                        // u=0 v=1 w=2 x=3 y=4 z=5
    G.addArc(0,1); G.addArc(0,3);        // u->v, u->x
    G.addArc(1,4);                       // v->y
    G.addArc(2,4); G.addArc(2,5);        // w->y, w->z
    G.addArc(3,1);                       // x->v
    G.addArc(4,3);                       // y->x
    G.addArc(5,5);                       // z->z  (a self-loop)

    DFS s(G);
    cout << "DFS, recursive\n";
    s.run();

    cout << "\ntimestamps\n  vertex :";
    for (int i = 0; i < 6; ++i) cout << "   " << NAME[i];
    cout << "\n  d/f    :";
    for (int i = 0; i < 6; ++i) cout << " " << s.d[i] << "/" << s.f[i];
    cout << "\n";

    cout << "\nparenthesization\n  " << s.parens << "\n";
```

Listing 10.4 (Continued) *Depth-first search, recursive, on the example of §10.8.4. It prints the trace, the timestamps, the parenthesization of §10.8.8, the forest, and the classification of every arc.*

```
    cout << "\ndepth-first forest (tree edges)\n  ";
    for (int v = 0; v < 6; ++v)
        if (s.pi[v] >= 0) cout << "(" << NAME[s.pi[v]] << "," << NAME[v] << ") ";
    cout << "\n  roots: ";
    for (int v = 0; v < 6; ++v) if (s.pi[v] < 0) cout << NAME[v] << " ";
    cout << "\n";

    cout << "\nedge classification\n";
    for (const string& line : s.classified) cout << "  " << line << "\n";

    cout << "\nvertices in order of finishing\n  ";
    for (int v : s.finishOrder) cout << NAME[v] << " ";
    cout << "\n";
    return 0;
}
```

Its output:

```
DFS, recursive
  d[u] = 1   u -> gray
    edge (u,v) is a tree edge
  d[v] = 2   v -> gray
    edge (v,y) is a tree edge
  d[y] = 3   y -> gray
    edge (y,x) is a tree edge
  d[x] = 4   x -> gray
    edge (x,v) is a back edge
  f[x] = 5   x -> black
  f[y] = 6   y -> black
  f[v] = 7   v -> black
    edge (u,x) is a forward edge
  f[u] = 8   u -> black
  d[w] = 9   w -> gray
    edge (w,y) is a cross edge
    edge (w,z) is a tree edge
  d[z] = 10   z -> gray
    edge (z,z) is a back edge
  f[z] = 11   z -> black
  f[w] = 12   w -> black

timestamps
  vertex :   u   v   w   x   y   z
  d/f    : 1/8 2/7 9/12 4/5 3/6 10/11

parenthesization
  (u (v (y (x x) y) v) u) (w (z z) w)

depth-first forest (tree edges)
  (u,v) (y,x) (v,y) (w,z)
  roots: u w

edge classification
  (u,v)  tree
  (v,y)  tree
  (y,x)  tree
  (x,v)  back
  (u,x)  forward
  (w,y)  cross
  (w,z)  tree
  (z,z)  back

vertices in order of finishing
  x y v u z w
```

Recursive and iterative agree on every timestamp, on every tree edge, and on the class of every arc — as they must, since they differ only in where the stack lives.

10.8.7 Why it costs $\Theta(V + E)$

The derivation, in four steps:

- *The loops in DFS take time $\Theta(V)$ each, excluding the time to execute DFS-Visit.* There are two such loops: the whitening loop and the loop that tries every vertex as a root. Each runs $|V|$ times doing constant work, ignoring the recursive calls.
- *DFS-Visit is called once for every vertex.* Why exactly once? Because it is invoked only on white vertices, and its **first action** is to paint the vertex gray. No vertex is ever repainted white. So the total number of calls is at most $|V|$; and since the outer loop tries every vertex, it is exactly $|V|$.
- *For each DFS-Visit a loop iterates over all* $\mathrm{Adj}[v]$ *— the total cost for DFS-Visit is $\Theta(E)$.* Each vertex's adjacency list is walked exactly once, inside its one call. So the total number of list

entries examined is $\sum_v |\mathrm{Adj}[v]|$, which is $|E|$ for a directed graph and $2|E|$ for an undirected one — the handshake lemma again.

- *The running time of DFS is $\Theta(V + E)$.*

So

$$T(n) = \underbrace{\Theta(V)}_{\text{two outer loops}} + \underbrace{\Theta(V)}_{|V|\text{ calls, constant overhead each}} + \underbrace{\Theta(E)}_{\text{walking all the lists}} = \Theta(V + E). \tag{10.11}$$

Note that Equation (10.11) is Θ, not O, whereas Equation (10.4) was only O: DFS always touches every vertex and every edge, so the bound is tight both ways. BFS from a single source is only $O(V + E)$, because if the graph is disconnected BFS may finish early having touched only part of it.

The technique in both derivations deserves a name, since it recurs throughout the book.

> **KEY IDEA — aggregate analysis.** We did not bound the cost of one iteration and multiply by the number of iterations; that would have given $O(V) \times O(V) = O(V^2)$, a correct but hopeless bound. Instead we bounded the **total** over all iterations, by observing that the inner loops together examine each adjacency-list entry exactly once. Charging the work to the *thing being processed* rather than to the *step doing the processing* is the whole trick, and Chapter 14 makes a formal method of it under the name *amortized analysis*.

10.8.8 The parenthesis theorem

Now the structure. It is best introduced with a picture that is worth more than any explanation: write "(" followed by u when u is discovered, and u followed by ")" when u finishes, in clock order. The result is:

Discovery and finish times have parenthesis structure. Represent discovery of u with left parenthesis "(u"; represent finishing of u with right parenthesis "u)". History of discoveries and finishings makes a well-formed expression (parentheses are properly nested).

Our example's history, read straight out of **Listing 10.4**'s output, is

$$(u\ (v\ (y\ (x\ \ x)\ y)\ v)\ u) \quad (w\ (z\ \ z)\ w)$$

and you can check by eye that every parenthesis closes in the right place. Nothing like $(u\ (v\ u)\ v)$ ever occurs.

For a second example we use a larger graph, and this one is worth working because it has two trees and all four edge classes. Its arcs are

$$s \to z,\ s \to w,\ z \to y,\ z \to w,\ y \to x,\ x \to z,\ w \to x,\ v \to w,\ v \to s,\ t \to v,\ t \to u,\ u \to t,\ u \to v.$$

Figure 10.25 is its timestamped drawing and **Figure 10.26** is the parenthesis structure beneath.

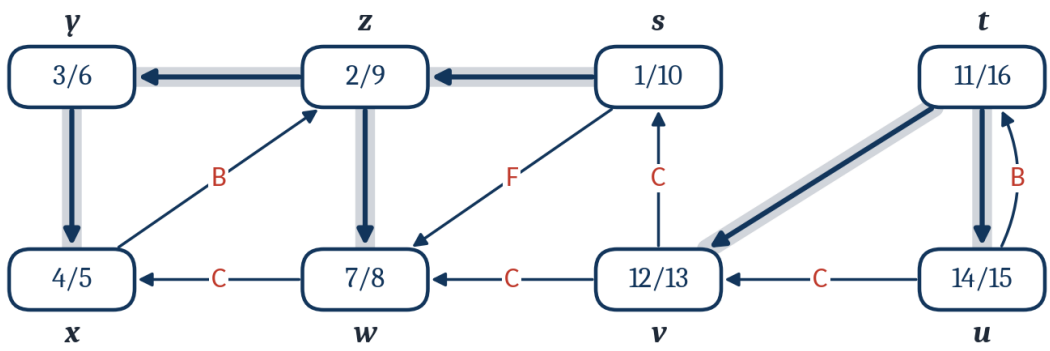


Figure 10.25 The larger DFS example, timestamped. Each vertex box holds d/f: s is $1/10$, z is $2/9$, y is $3/6$, x is $4/5$, w is $7/8$, t is $11/16$, v is $12/13$, u is $14/15$. The thick gray arrows are tree edges; the thin ones are labeled **B** (back), **F** (forward) and **C** (cross). This one figure carries the timestamps, the forest and the complete edge classification at once.

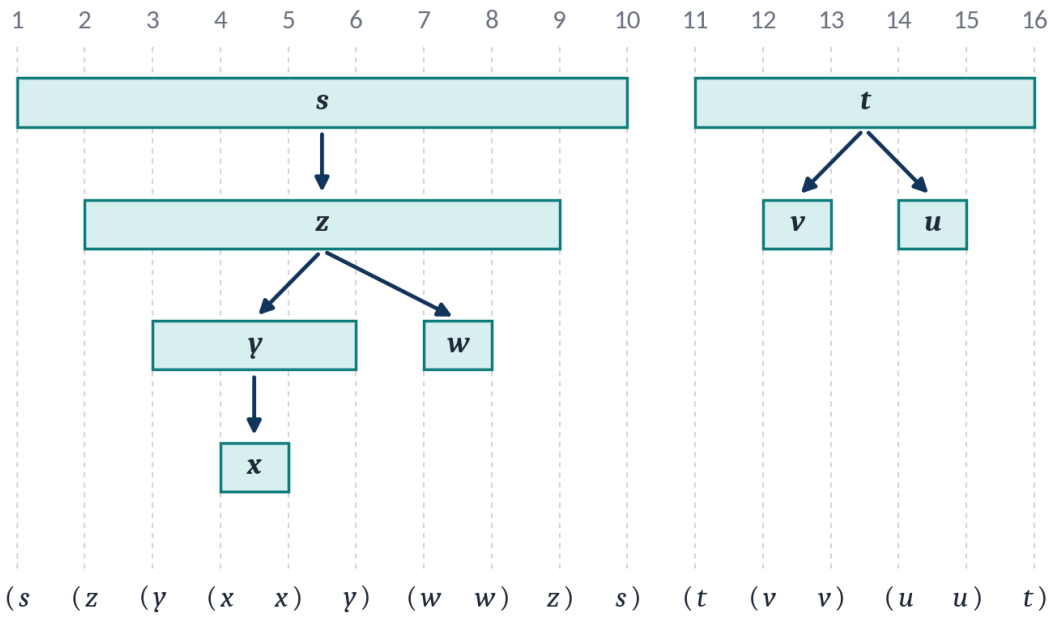


Figure 10.26 The parenthesis structure of **Figure 10.25**. Above, the depth-first forest redrawn with tree edges pointing downward. Below, the clock from 1 to 16 with one horizontal bar per vertex, spanning $[d, f]$, and the parenthesization written along the bottom: $(s\ (z\ (y\ (x\ x)\ y)\ (w\ w)\ z)\ s)\ (t\ (v\ v)\ (u\ u)\ t)$. Every pair of bars is either nested or disjoint; none overlap partially.

Listing 10.5 runs this second example, to confirm that our reading of the picture is the same as the picture.

Listing 10.5 The second DFS example run in full, to check the timestamps and the parenthesization of **Figure 10.25** and **Figure 10.26**.

```
// DFS on the second directed example, the one whose parenthesis
// structure is drawn in Section 10.8.8:
//   V = {s,t,u,v,w,x,y,z}
//   E = {s->z, s->w, z->y, z->w, y->x, x->z, w->x, v->w, v->s, t->v, t->u,
//        u->t, u->v}
// Vertices are visited in the order s, t, u, v, w, x, y, z of the main loop.
#include <iostream>
#include <vector>
#include <string>
using namespace std;

enum Color { WHITE, GRAY, BLACK };
static const string NAME = "stuvwxyz";     // s=0 t=1 u=2 v=3 w=4 x=5 y=6 z=7

struct DFS {
    vector<vector<int>> adj;
    vector<Color> color;
    vector<int> d, f, pi;
    int time = 0;
    string parens;
    vector<pair<string,string>> edges;

    explicit DFS(vector<vector<int>> a)
        : adj(move(a)), color(adj.size(), WHITE),
          d(adj.size(),0), f(adj.size(),0), pi(adj.size(),-1) {}

    void run() { for (size_t u = 0; u < adj.size(); ++u) if (color[u]==WHITE)
visit((int)u); }

    void visit(int u) {
        color[u] = GRAY; d[u] = ++time;
        parens += "(" + string(1,NAME[u]) + " ";
        for (int v : adj[u]) {
            string kind = color[v]==WHITE ? "tree"
                        : color[v]==GRAY  ? "back"
                        : d[u] < d[v]      ? "forward" : "cross";
            edges.push_back({string(1,NAME[u]) + "->" + string(1,NAME[v]), kind});
            if (color[v]==WHITE) { pi[v]=u; visit(v); }
        }
        color[u] = BLACK; f[u] = ++time;
        parens += string(1,NAME[u]) + ") ";
    }
};
```

Listing 10.5 (Continued) *The second DFS example run in full, to check the timestamps and the parenthesization of Figure 10.25 and Figure 10.26.*

```
int main() {
    vector<vector<int>> adj(8);
    adj[0] = {7,4};      // s -> z, w
    adj[7] = {6,4};      // z -> y, w
    adj[6] = {5};        // y -> x
    adj[5] = {7};        // x -> z
    adj[4] = {5};        // w -> x
    adj[3] = {4,0};      // v -> w, s
    adj[1] = {3,2};      // t -> v, u
    adj[2] = {1,3};      // u -> t, v

    DFS s(adj);
    s.run();

    cout << "timestamps\n  vertex :";
    for (int i = 0; i < 8; ++i) cout << "     " << NAME[i];
    cout << "\n  d/f    :";
    for (int i = 0; i < 8; ++i) cout << " " << s.d[i] << "/" << s.f[i]
                                    << (s.f[i] < 10 ? "  " : "");
    cout << "\n\nparenthesization\n  " << s.parens << "\n";
    cout << "\nedge classification\n";
    for (auto& e : s.edges) cout << "  " << e.first << "  " << e.second << "\n";
    return 0;
}
```

Its output:

```
timestamps
  vertex :     s     t     u     v     w     x     y     z
  d/f    : 1/10 11/16 14/15 12/13 7/8   4/5   3/6   2/9

parenthesization
  (s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

edge classification
  s->z  tree
  z->y  tree
  y->x  tree
  x->z  back
  z->w  tree
  w->x  cross
  s->w  forward
  t->v  tree
  v->w  cross
  v->s  cross
  t->u  tree
  u->t  back
  u->v  cross
```

Every timestamp agrees with **Figure 10.25**, and the parenthesization is character-for-character the one written along the bottom of **Figure 10.26**. Now the theorem.

> **PARENTHESIS THEOREM (Theorem 10.6).** *In any depth-first search of any graph, for any two vertices u and v, exactly one of the following three holds:*

1. *the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are **disjoint**, and neither u nor v is a descendant of the other in the depth-first forest;*
2. *$[d[u], f[u]]$ is **contained entirely within** $[d[v], f[v]]$, and u is a descendant of v;*
3. *$[d[v], f[v]]$ is **contained entirely within** $[d[u], f[u]]$, and v is a descendant of u.*

> *In particular, the intervals never **partially overlap**.*

Proof. Assume without loss of generality that $d[u] < d[v]$ — that is, u was discovered first. (If $d[v] < d[u]$, swap the names; they cannot be equal, since the clock never repeats a value.) There are two cases.

Case A: $d[v] < f[u]$. So v was discovered while u was still gray. A vertex is gray exactly while its own DFS-Visit call is on the stack, so DFS-Visit(v) began during DFS-Visit(u), which means DFS-Visit(v) is nested inside DFS-Visit(u). And a nested call must return before its caller returns — this is a property of the call stack itself, not of graphs. So v finishes before u does: $f[v] < f[u]$. Hence

$$d[u] < d[v] < f[v] < f[u], \tag{10.12}$$

which is exactly case 3: v's interval sits strictly inside u's, and v was discovered from a chain of gray vertices starting at u, so v is a descendant of u.

Case B: $f[u] < d[v]$. Then u finished before v was discovered, so

$$d[u] < f[u] < d[v] < f[v],$$

and the two intervals are disjoint. Neither can be a descendant of the other: for v to be a descendant of u, DFS-Visit(v) would have to be nested inside DFS-Visit(u), which requires $d[v] < f[u]$ — false here; and for u to be a descendant of v we would need $d[v] < d[u]$, also false. This is case 1.

Those are the only two cases, because $d[v] \neq f[u]$ (all $2|V|$ stamps are distinct) and so either $d[v] < f[u]$ or $f[u] < d[v]$. A *partial* overlap would require something like $d[u] < d[v] < f[u] < f[v]$: but we have just shown that $d[v] < f[u]$ forces $f[v] < f[u]$, so this arrangement cannot occur. ■

The intuition behind the proof is worth keeping: *any two intervals are either disjoint or enclosed; overlapping intervals would mean finishing an ancestor before finishing a descendant, or starting a descendant without starting an ancestor.* Both are impossible on a call stack.

Corollary 10.7 (nesting of descendants' intervals). *Vertex v is a proper descendant of vertex u in the depth-first forest of G if and only if $d[u] < d[v] < f[v] < f[u]$, the nesting of* Equation (10.12)*.*

This is the practical form of the theorem, and it turns a question about *tree shape* into a comparison of *four integers*. Test it on **Table 10.9**: is x a descendant of u? $d[u] = 1 < d[x] = 4 < f[x] = 5 < f[u] = 8$. Yes. Is y a descendant of w? $d[w] = 9$ and $d[y] = 3$, so $d[w] > d[y]$ and the chain fails immediately. No — and indeed they are in different trees.

10.8.9 The white-path theorem

One more structural result. It is the reason DFS finds everything it should, and it is the engine of the DAG theorem in §10.9.5.

> **WHITE-PATH THEOREM (Theorem 10.8).** *In a depth-first forest of G, vertex v is a descendant of vertex u if and only if, at the instant $d[u]$ when u is discovered, there is a path from u to v consisting* ***entirely of white vertices*** *(apart from u itself, which turns gray at that instant).*

Proof sketch, forward. If v is a descendant of u, then the tree path from u down to v exists. Every vertex on it was discovered after u — by Corollary 10.7, $d[u] < d[w]$ for each such w — so at time $d[u]$ every one of them was still white. That tree path is a white path.

Proof, backward. Suppose at time $d[u]$ there is a white path $u \rightsquigarrow v$ but v does not become a descendant of u. Among the vertices on that path that fail to become descendants of u, let w be the one closest to u, and let t be the vertex just before w on the path, so t *does* become a descendant of u (or $t = u$). By Corollary 10.7, $f[t] \leq f[u]$.

Now w is white at time $d[u]$, so $d[u] < d[w]$. And w must be discovered before t finishes, because the edge (t, w) is examined during DFS-Visit(t): either w is still white then, in which case t becomes its parent and w *is* a descendant of u — contradiction — or w has already been discovered, in which case $d[w] < f[t]$. Either way $d[u] < d[w] < f[t] \leq f[u]$, so $d[u] < d[w] < f[u]$, and by the parenthesis theorem w's interval is nested inside u's, making w a descendant of u. Contradiction. ■

The theorem reads better as a slogan: **a depth-first search from u reaches everything that is reachable from u through territory not yet explored.** It cannot be blocked by a vertex it has not seen, only by one it already has.

10.8.10 Classifying the edges

We have used the four edge classes informally; now they get definitions, with the color test attached to each.

> **DEFINITION.** Fix a depth-first search of G, producing the forest G_π. Every edge (u, v) of G falls into exactly one of four classes, decided by the color of v at the moment the edge (u, v) is examined from the gray vertex u:

- a **tree edge** — v is WHITE — is an edge of G_π; it discovers a new vertex;
- a **back edge** — v is GRAY — goes from u to an ancestor of u in G_π (a self-loop counts as a back edge);
- a **forward edge** — v is BLACK and $d[u] < d[v]$ — is a non-tree edge from u to a proper descendant of u;
- a **cross edge** — v is BLACK and $d[u] > d[v]$ — is everything else: between two subtrees of the same tree, or between two different trees.

In short, for the first two: *tree edge (gray to white) — encounter new vertices; back edge (gray to gray) — from descendant to ancestor*, and for the second two *forward edge (gray to black) — from ancestor to descendant; cross edge (gray to black) — remainder, between trees or subtrees*. Note that forward and cross edges are distinguished by the same color test, so the timestamps must be consulted to tell them apart: a forward edge points *down* into already-finished territory below you, a cross edge points *sideways* into already-finished territory beside you.

Figure 10.27 is the best picture of this, because it redraws the whole graph so that tree edges hang downward; once the picture is drawn that way the words *back*, *forward* and *cross* describe exactly what the eye sees.

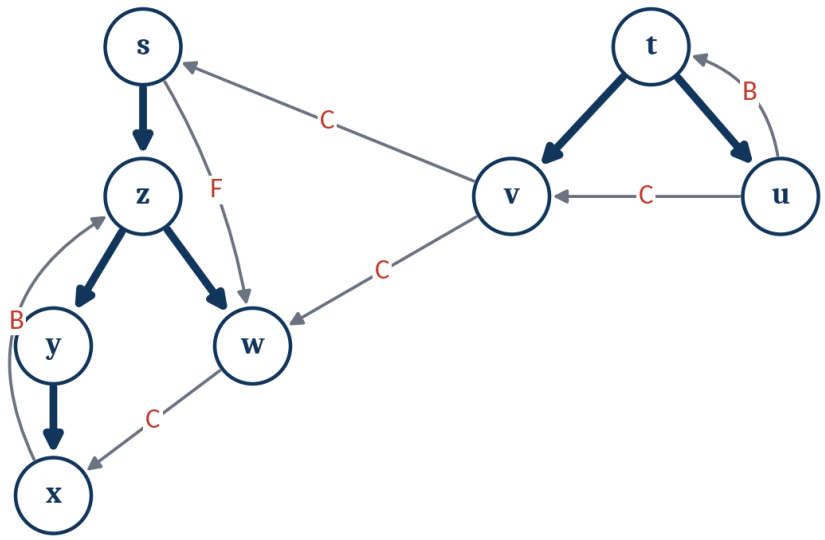


Figure 10.27 The graph of **Figure 10.25** redrawn as its depth-first forest, with tree edges thick and downward. Now the classification is visual: $x \to z$ and $u \to t$ curve back **up** to an ancestor (**B**); $s \to w$ points **down** past an intermediate level to a descendant (**F**); $w \to x$, $v \to w$, $v \to s$ and $u \to v$ go **sideways** between branches or between the two trees (**C**).

Two practical remarks.

Which classes matter? To put it plainly: *tree and back edges are important; most algorithms do not distinguish between forward and cross edges.* Tree edges give you the forest, and hence the structure. Back edges give you the cycles, and hence acyclicity, strong connectivity, articulation points and bridges. Forward and cross edges are usually ignored.

Which classes can even occur? This depends on whether the graph is directed, and the answer is a genuinely useful theorem.

THEOREM 10.9. *In a depth-first search of an* ***undirected*** *graph G, every edge is either a tree edge or a back edge. Forward and cross edges never occur.*

Proof. Let (u, v) be an edge, and suppose without loss of generality $d[u] < d[v]$; so u is discovered first. An edge is classified the first time it is examined, from whichever end that happens, so we ask which end comes first.

Step 1: v is a descendant of u. Because the graph is undirected, $v \in \text{Adj}[u]$, so DFS-Visit(u) examines the edge before u finishes. By then v has certainly been discovered — if it was still white, it is discovered at that very moment. So $d[u] < d[v] < f[u]$, and by the parenthesis theorem (Case A of its proof) v is a descendant of u, with $f[v] < f[u]$.

Step 2: if the edge is first examined from u's side, it is a tree edge. At that moment DFS-Visit(u) is the call currently running, so u is the deepest gray vertex. If v were gray, it would be a gray descendant of u, deeper than u — impossible. If v were black, DFS-Visit(v) would already have scanned all of Adj[v], which contains u, so the edge would already have been examined from v's side — contradicting "first". So v is white, and (u, v) is a tree edge.

Step 3: if the edge is first examined from v's side, it is a back edge. That examination happens during DFS-Visit(v), and by Step 1 that call is nested inside DFS-Visit(u). So u is still gray, and an edge into a gray vertex is a back edge.

Either way the edge is a tree edge or a back edge; a forward or cross edge would need a black head on the first examination, and we have just shown that cannot happen. ■

The practical consequence is the one-line cycle test for undirected graphs in §10.9.2: since the only non-tree edges are back edges, *any* non-tree edge means a cycle.

Table 10.11 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

10.9 Six Things the Two Searches Can Do

We now have two linear-time searches and a pile of theorems about them. This section spends the capital. Six problems, none of which mentions searching in its statement, all of which are solved in $O(V + E)$ by a search with about five lines added.

10.9.1 Connected components

Problem. Given an undirected graph, how many connected components does it have, and which component is each vertex in?

Idea. By §10.4.8, a search started inside a component can never leave it, because no edge crosses between components. So: start a search at vertex 0; everything it reaches is component 0. Find the lowest-numbered vertex it did *not* reach; start a search there; everything *that* reaches is component 1. Repeat until no vertex is left unlabeled.

Either search works. BFS is used below because the component label replaces the `color` array entirely: a vertex with no label is white, and a vertex with a label has been seen.

Correctness. Two claims. *Everything labeled c really is in one component:* every vertex the search from s reaches has a path from s, so all of them are mutually reachable through s. *Everything in s's component really gets label c:* by Theorem 10.4, BFS discovers every vertex reachable from s, and in an undirected graph "reachable from s" is exactly "in s's component".

Cost. The outer loop runs $|V|$ times, and the searches together enqueue each vertex at most once and scan each adjacency list at most once — because a vertex already labeled is skipped. So the total is $O(V + E)$, not $O(V \cdot (V + E))$. This is aggregate analysis again: the work is charged to the vertices and edges, not to the iterations of the outer loop.

Listing 10.6 labels the graph of **Figure 10.9**, which has three components.

Listing 10.6 Connected components by repeated breadth-first search, on a graph shaped like **Figure 10.9**. Once `comp[]` is built, "are u and v connected?" becomes a single integer comparison.

```
// Connected components of an undirected graph, by repeated BFS.
// The graph has the shape of the three-component figure of Section 10.4.8:
// a triangle, a star, and a five-vertex piece.
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

// comp[v] = index of the component containing v; returns the number of components
int components(const vector<vector<int>>& adj, vector<int>& comp) {
    int n = (int)adj.size();
    comp.assign(n, -1);
    int c = 0;
    for (int s = 0; s < n; ++s) {
        if (comp[s] != -1) continue;         // already placed by an earlier search
        queue<int> Q; Q.push(s); comp[s] = c;
        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            for (int v : adj[u])
                if (comp[v] == -1) { comp[v] = c; Q.push(v); }
        }
        ++c;                                  // s exhausted one whole component
    }
    return c;
}

int main() {
    const int n = 12;
    vector<vector<int>> adj(n);
    auto add = [&](int u, int v) { adj[u].push_back(v); adj[v].push_back(u); };
    add(0,1); add(1,2); add(2,0);                                   // triangle
    add(3,4); add(3,5); add(3,6);                                   // star
    add(7,8); add(7,9); add(7,10); add(8,9); add(9,10); add(9,11); add(10,11);

    vector<int> comp;
    int k = components(adj, comp);
    cout << "|V| = " << n << ", components = " << k << "\n\n";

    vector<vector<int>> members(k);
    for (int v = 0; v < n; ++v) members[comp[v]].push_back(v);
    for (int c = 0; c < k; ++c) {
        cout << "  component " << c << " (" << members[c].size() << " vertices): ";
        for (int v : members[c]) cout << v << " ";
        cout << "\n";
    }
```

Listing 10.6 (Continued) *Connected components by repeated breadth-first search, on a graph shaped like Figure 10.9. Once comp[] is built, "are u and v connected?" becomes a single integer comparison.*

```
    cout << "\nsame-component queries, answered in O(1) from comp[]\n";
    const int q[4][2] = {{0,2},{0,4},{4,6},{8,11}};
    for (auto& p : q)
        cout << "  " << p[0] << " and " << p[1] << ": "
             << (comp[p[0]] == comp[p[1]] ? "connected" : "not connected") << "\n";
    return 0;
}
```

Its output:

```
|V| = 12, components = 3

  component 0 (3 vertices): 0 1 2
  component 1 (4 vertices): 3 4 5 6
  component 2 (5 vertices): 7 8 9 10 11

same-component queries, answered in O(1) from comp[]
  0 and 2: connected
  0 and 4: not connected
  4 and 6: connected
  8 and 11: connected
```

Three components of sizes 3, 4 and 5 — the triangle, the star and the five-vertex piece of **Figure 10.9**. Note what the last block buys you: after one $O(V + E)$ pass, every future connectivity query costs one comparison. That is the standard pattern for a preprocessing algorithm, and it is why a component label is worth computing even if you only expect a few queries.

And connectivity for free. A graph is connected exactly when the count is 1. So `components(adj, comp) == 1` is the connectivity test, and it costs one linear pass.

10.9.2 Cycle detection

Problem. Does this graph contain a cycle?

Here the undirected and directed cases genuinely differ, and the difference is instructive.

Undirected. By Theorem 10.9, a DFS of an undirected graph produces only tree edges and back edges. A back edge means a cycle: the edge (u, v) plus the tree path from v down to u closes one. So the graph has a cycle if and only if the DFS finds a back edge — that is, if and only if it ever examines an edge into a non-white vertex.

But there is a catch, and it is the classic beginner's trap. When the search is at x, having arrived from y, the edge (x, y) is in $\mathrm{Adj}[x]$ and y is gray. That is not a cycle; it is the edge we walked in on, seen from the other end. So the test must ignore the edge back to the parent — **exactly once**. Ignoring it once and not more is what makes the program also detect a *multigraph* cycle: two parallel edges between x and y genuinely are a cycle of length 2, and the second encounter with the parent must therefore be reported. **Listing 10.7** does this with one line: `if (v == p) { p = -2; continue; }` — after skipping the parent once, `p` is set to an impossible vertex number so no later neighbor can match it.

Directed. Theorem 10.9 does not apply, so all four classes occur, and only *back* edges indicate a cycle. Forward and cross edges do not: $u \to x$ in our example is a forward edge and u, x lie on no cycle together. The test is therefore not "non-white" but precisely "**gray**". And gray means, by §10.8.1, *an ancestor of where I am standing* — so an arc into a gray vertex closes a directed cycle through the chain of gray vertices, and nothing else does.

> **WARNING.** The single most common bug in directed cycle detection is testing `visited[v]` instead of `gray[v]`. A `bool visited[]` cannot distinguish gray from black, so it reports cross and forward edges as cycles. The graph $u \to v, u \to w, v \to w$ has no cycle at all, but a `visited`-based test flags $v \to w$ and reports one. This is exactly the situation the three-color scheme exists for.

Cost. One DFS: $O(V + E)$, and it can stop early the moment it finds a back edge.

A shortcut for the undirected case. If the graph is connected, it is a tree exactly when $|E| = |V| - 1$ (§10.4.11), so counting edges answers the question without any search. More generally,

an undirected graph with k components is a forest exactly when $|E| = |V| - k$, and if $|E| > |V| - k$ it has a cycle. This costs one pass to count components and is worth knowing; but it tells you only *whether*, not *where*, and it says nothing in the directed case.

Listing 10.7 Cycle detection for both undirected and directed graphs, tested on seven graphs including the chapter's own examples. The undirected test accepts any non-white neighbor other than the parent; the directed test insists on **gray**.

```
// Cycle detection, both flavors.
//   undirected: a back edge that is not the edge we arrived on
//   directed:   an edge into a GRAY vertex
#include <iostream>
#include <vector>
#include <string>
using namespace std;

enum Color { WHITE, GRAY, BLACK };

// ---------- undirected ----------
struct UndirectedCycle {
    vector<vector<int>> adj;
    vector<Color> color;
    vector<int> parent;
    int a = -1, b = -1;                      // the two ends of the offending edge

    explicit UndirectedCycle(vector<vector<int>> g)
        : adj(move(g)), color(adj.size(), WHITE), parent(adj.size(), -1) {}

    bool hasCycle() {
        for (size_t s = 0; s < adj.size(); ++s)
            if (color[s] == WHITE && visit((int)s, -1)) return true;
        return false;
    }
    bool visit(int u, int p) {
        color[u] = GRAY; parent[u] = p;
        for (int v : adj[u]) {
            if (v == p) { p = -2; continue; }   // skip the edge we came in on, once
            if (color[v] != WHITE) { a = u; b = v; return true; }
            if (visit(v, u)) return true;
        }
        color[u] = BLACK;
        return false;
    }
};

// ---------- directed ----------
struct DirectedCycle {
    vector<vector<int>> adj;
    vector<Color> color;
    int a = -1, b = -1;

    explicit DirectedCycle(vector<vector<int>> g)
        : adj(move(g)), color(adj.size(), WHITE) {}
```

Listing 10.7 (Continued) *Cycle detection for both undirected and directed graphs, tested on seven graphs including the chapter's own examples. The undirected test accepts any non-white neighbor other than the parent; the directed test insists on gray.*

```
    bool hasCycle() {
        for (size_t s = 0; s < adj.size(); ++s)
            if (color[s] == WHITE && visit((int)s)) return true;
        return false;
    }
    bool visit(int u) {
        color[u] = GRAY;
        for (int v : adj[u]) {
            if (color[v] == GRAY) { a = u; b = v; return true; }  // back edge
            if (color[v] == WHITE && visit(v)) return true;
        }
        color[u] = BLACK;
        return false;
    }
};

static vector<vector<int>> undirectedFrom(int n, const vector<pair<int,int>>& E) {
    vector<vector<int>> adj(n);
    for (auto e : E) { adj[e.first].push_back(e.second); adj[e.second].push_back(e.first);
}
    return adj;
}
static vector<vector<int>> directedFrom(int n, const vector<pair<int,int>>& E) {
    vector<vector<int>> adj(n);
    for (auto e : E) adj[e.first].push_back(e.second);
    return adj;
}

int main() {
    cout << "undirected graphs\n";
    {   // the chapter's graph: V={a..e}, E={ab,ac,ad,be,cd,ce,de}
        UndirectedCycle t(undirectedFrom(5, {{0,1},{0,2},{0,3},{1,4},{2,3},{2,4},{3,4}}));
        cout << "  a-e graph of the chapter : " << (t.hasCycle() ? "has a cycle" :
"acyclic");
        if (t.a >= 0) cout << "  (back edge " << char('a'+t.a) << "-" << char('a'+t.b) <<
")";
        cout << "\n";
    }
    {   // a tree on 5 vertices
        UndirectedCycle t(undirectedFrom(5, {{0,1},{0,2},{1,3},{1,4}}));
        cout << "  tree on 5 vertices      : " << (t.hasCycle() ? "has a cycle" :
"acyclic") << "\n";
    }
    {   // a forest: two trees, |E| = |V| - 2
        UndirectedCycle t(undirectedFrom(6, {{0,1},{1,2},{3,4},{4,5}}));
        cout << "  forest of two paths     : " << (t.hasCycle() ? "has a cycle" :
"acyclic") << "\n";
    }
    {   // one triangle plus an isolated vertex
        UndirectedCycle t(undirectedFrom(4, {{0,1},{1,2},{2,0}}));
        cout << "  triangle + isolate      : " << (t.hasCycle() ? "has a cycle" :
"acyclic") << "\n";
```

Listing 10.7 (Continued) *Cycle detection for both undirected and directed graphs, tested on seven graphs including the chapter's own examples. The undirected test accepts any non-white neighbor other than the parent; the directed test insists on gray.*

```
    }

    cout << "\ndirected graphs\n";
    {   // the DFS example of the chapter: u->v,u->x,v->y,w->y,w->z,x->v,y->x,z->z
        DirectedCycle t(directedFrom(6,
{{0,1},{0,3},{1,4},{2,4},{2,5},{3,1},{4,3},{5,5}}));
        cout << "  DFS example of the chapter: " << (t.hasCycle() ? "has a directed cycle"
: "is a DAG");
        if (t.a >= 0) cout << "  (back edge " << "uvwxyz"[t.a] << "->" << "uvwxyz"[t.b] <<
")";
        cout << "\n";
    }
    {   // the dressing DAG, with the vertex numbering used later in the chapter
        DirectedCycle t(directedFrom(9,
{{0,1},{0,3},{1,2},{3,2},{5,6},{5,7},{6,7},{6,3},{8,7}}));
        cout << "  dressing DAG              : " << (t.hasCycle() ? "has a directed cycle"
: "is a DAG") << "\n";
    }
    {   // the same DAG with one extra arc jacket -> shirt, which closes a cycle
        DirectedCycle t(directedFrom(9,
{{0,1},{0,3},{1,2},{3,2},{5,6},{5,7},{6,7},{6,3},{8,7},{2,0}}));
        cout << "  dressing DAG + jacket->shirt: " << (t.hasCycle() ? "has a directed
cycle" : "is a DAG") << "\n";
    }
    return 0;
}
```

Its output:

```
undirected graphs
  a-e graph of the chapter : has a cycle  (back edge c-a)
  tree on 5 vertices       : acyclic
  forest of two paths      : acyclic
  triangle + isolate       : has a cycle

directed graphs
  DFS example of the chapter: has a directed cycle  (back edge x->v)
  dressing DAG              : is a DAG
  dressing DAG + jacket->shirt: has a directed cycle
```

Each line is checkable by hand. The chapter's graph has the triangle $a\ c\ d$ of **Figure 10.7**, and the back edge reported, c–a, is one of its three edges. The DFS example's reported back edge, $x \to v$, is exactly the one labeled **B** in **Figure 10.22**. The five-vertex tree and the two-path forest are acyclic, as **Figure 10.10** would have us expect. And adding one arc to a DAG destroys its acyclicity — the topic of §10.9.5.

10.9.3 Testing bipartiteness

Problem. Can the vertices be split into two sides so that every edge crosses between them (§10.4.13)?

Idea. Run BFS and color each vertex by the parity of its distance: side 0 if $d[v]$ is even, side 1 if odd. By Corollary 10.5, every edge joins levels differing by at most one — so a *tree* edge always crosses, since consecutive levels have opposite parity. The only possible failure is an edge joining two vertices at the **same** level, which would join two vertices of the same parity. So: do the BFS, and whenever an edge leads to an already-colored vertex, check that the colors differ. If one ever does not, the graph is not bipartite; if none ever does, the assignment we built is a valid bipartition, which proves it is.

Why the failure really is a failure. Suppose the edge (u, v) joins two vertices with $d[u] = d[v] = k$. Follow the tree path from u up to the root s, and from v up to s; let z be the first vertex they share. The two path segments $z \rightsquigarrow u$ and $z \rightsquigarrow v$ have the same length, say ℓ. Then $z \rightsquigarrow u$, the edge (u, v), and $v \rightsquigarrow z$ reversed form a closed walk of length $2\ell + 1$ — **odd**. It contains an odd cycle. And by §10.4.13 a graph with an odd cycle is not bipartite. So the algorithm's "no" is justified, not merely a failure to find a coloring.

Disconnected graphs. Each component is colored independently, starting from its own source. That is correct, because no edge crosses between components, so no constraint links them. This is why the loop over all vertices is needed, not just one BFS.

Cost. One BFS per component: $O(V + E)$ in total.

Listing 10.8 Bipartiteness testing by two-coloring during a BFS. On success it prints the two sides, which is the certificate; on failure it names the edge that could not be colored, which points at an odd cycle.

```
// Bipartiteness testing by BFS two-coloring.
// A graph is bipartite exactly when it can be 2-colored so that no edge
// joins two vertices of the same color; BFS assigns color d[v] mod 2.
#include <iostream>
#include <vector>
#include <queue>
#include <string>
using namespace std;

// Returns true and fills side[] with 0/1 if bipartite; otherwise returns false
// and leaves in bad[] the two ends of an edge that cannot be colored.
bool bipartite(const vector<vector<int>>& adj, vector<int>& side, pair<int,int>& bad) {
    int n = (int)adj.size();
    side.assign(n, -1);
    for (int s = 0; s < n; ++s) {
        if (side[s] != -1) continue;          // this component already done
        side[s] = 0;
        queue<int> Q; Q.push(s);
        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            for (int v : adj[u]) {
                if (side[v] == -1) { side[v] = 1 - side[u]; Q.push(v); }
                else if (side[v] == side[u]) { bad = {u, v}; return false; }
            }
        }
    }
    return true;
}

static vector<vector<int>> build(int n, const vector<pair<int,int>>& E) {
    vector<vector<int>> adj(n);
    for (auto e : E) { adj[e.first].push_back(e.second); adj[e.second].push_back(e.first);
}
    return adj;
}

static void report(const string& what, int n, const vector<pair<int,int>>& E) {
    vector<int> side; pair<int,int> bad{-1,-1};
    bool ok = bipartite(build(n, E), side, bad);
    cout << "  " << what << ": ";
    if (!ok) {
        cout << "NOT bipartite (edge " << bad.first << "-" << bad.second
             << " joins two vertices of the same side)\n";
        return;
    }
    cout << "bipartite; side 0 = {";
    bool first = true;
```

Listing 10.8 (Continued) *Bipartiteness testing by two-coloring during a BFS. On success it prints the two sides, which is the certificate; on failure it names the edge that could not be colored, which points at an odd cycle.*

```
    for (int v = 0; v < n; ++v) if (side[v]==0) { cout << (first?"":",") << v; first=false;
}
    cout << "}, side 1 = {";
    first = true;
    for (int v = 0; v < n; ++v) if (side[v]==1) { cout << (first?"":",") << v; first=false;
}
    cout << "}\n";
}

int main() {
    cout << "bipartiteness tests\n";
    report("even cycle C6      ", 6, {{0,1},{1,2},{2,3},{3,4},{4,5},{5,0}});
    report("odd cycle C5       ", 5, {{0,1},{1,2},{2,3},{3,4},{4,0}});
    report("path P4            ", 4, {{0,1},{1,2},{2,3}});
    report("complete K_{2,3}   ", 5, {{0,2},{0,3},{0,4},{1,2},{1,3},{1,4}});
    report("K4 (complete, n=4) ", 4, {{0,1},{0,2},{0,3},{1,2},{1,3},{2,3}});
    report("chapter graph a..e ", 5, {{0,1},{0,2},{0,3},{1,4},{2,3},{2,4},{3,4}});
    report("two disjoint edges ", 4, {{0,1},{2,3}});
    return 0;
}
```

Its output:

```
bipartiteness tests
  even cycle C6      : bipartite; side 0 = {0,2,4}, side 1 = {1,3,5}
  odd cycle C5       : NOT bipartite (edge 2-3 joins two vertices of the same side)
  path P4            : bipartite; side 0 = {0,2}, side 1 = {1,3}
  complete K_{2,3}   : bipartite; side 0 = {0,1}, side 1 = {2,3,4}
  K4 (complete, n=4) : NOT bipartite (edge 1-2 joins two vertices of the same side)
  chapter graph a..e : NOT bipartite (edge 2-3 joins two vertices of the same side)
  two disjoint edges : bipartite; side 0 = {0,2}, side 1 = {1,3}
```

Seven answers, all matching §10.4.13's hand analysis. The even cycle splits by parity; the odd cycle cannot; $K_{2,3}$ splits into its two intended sides; K_4 cannot, because any three of its four vertices form a triangle; and the chapter's own graph fails at the edge 2–3, which is c–d — one edge of the triangle $a\ c\ d$, exactly as predicted.

10.9.4 Shortest paths in an unweighted graph

This is not so much an application as the statement of what BFS already is, but it deserves its own heading because it is the most-used graph algorithm in the world.

Problem. Given an undirected or directed graph with no edge weights and a source s, find for every vertex v the fewest edges on any path from s to v, and one such path.

Solution. BFS from s. Theorem 10.4 says $d[v] = \delta(s, v)$; §10.7.8 says the π array encodes a shortest path to every reachable vertex. That is the whole answer, already computed and already printed by **Listing 10.2**.

Three refinements worth knowing.

Single target. If you want only the distance to one target t, stop the moment t is dequeued. The answer is already final, because vertices are dequeued in non-decreasing order of d (corollary to Lemma 10.3), so nothing discovered later can improve it. This can be a very large saving: a road network may have millions of vertices and the target may be three hops away.

Multiple sources. If you want, for every vertex, the distance to the *nearest* member of a set S of sources, put all of S into the queue at the start, each with $d = 0$. Everything in the proof of

Theorem 10.4 goes through unchanged with $\delta(S, v) = \min_{s \in S} \delta(s, v)$ in place of $\delta(s, v)$; the reason is that the initial queue still satisfies Lemma 10.3, since all its entries have the same d. One BFS, $O(V + E)$ — not $|S|$ separate searches.

Directed graphs. BFS is correct on directed graphs word for word; nothing in §10.7.9 used symmetry. The only difference is that $\mathrm{Adj}[u]$ now means the heads of arcs leaving u, so the distances are along arcs in their proper direction, and $\delta(s, v)$ need not equal $\delta(v, s)$.

What BFS cannot do. Repeating the warning of §10.4.14 because it is the most costly misunderstanding in this chapter: if the edges carry lengths, BFS is wrong. Use Dijkstra's algorithm (Chapter 13), which is BFS with the queue replaced by a priority queue. There is one useful exception: if every weight is the same constant c, then the fewest-edges path is also the least-weight path, and its weight is $c \cdot d[v]$. And a second, cheap trick for weights in $\{0,1\}$: use a double-ended queue, pushing 0-weight neighbors to the front and 1-weight neighbors to the back. That is *0-1 BFS*, it is still $O(V + E)$, and it is worth remembering for contest problems.

10.9.5 Directed acyclic graphs

DEFINITION. A **directed acyclic graph** (DAG) is a directed graph with no directed cycle.

Figure 10.28 illustrates two DAGs, and the sentence to remember about what DAGs are for is this: *often used to indicate precedences among events, i.e., event a must happen before b. An example would be a parallel code execution.*

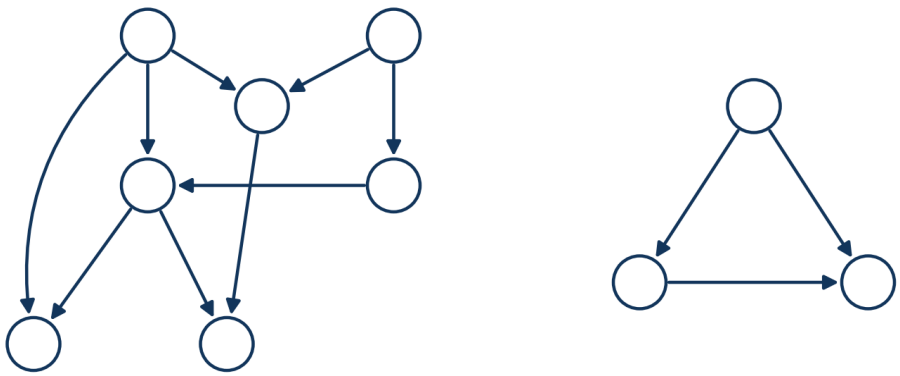


Figure 10.28 Two directed acyclic graphs. In each, follow the arrows from anywhere and you can never return to where you started: there is no directed cycle. A DAG is the natural model of a set of precedences — "this before that" — and an acyclic set of precedences is exactly a satisfiable one.

Almost every "this must come before that" situation is a DAG, or ought to be: the tasks of a build system, the courses of a degree with their prerequisites, the fields of a spreadsheet with their formulas, the stages of a manufacturing process, the version history of a source repository, the layers of a neural network. When such a graph turns out *not* to be acyclic, the specification is broken, and saying so is a service.

The characterization that makes all of this computable is the **DAG theorem**, and its proof is a lovely use of the white-path theorem.

DAG THEOREM (Theorem 10.10). *A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.*

Proof. There are two directions, and we prove both.

(⇒, in contrapositive form: a back edge implies a cycle.) Suppose the DFS produces a back edge (u, v). By the definition in §10.8.10, v is an ancestor of u in the depth-first forest. So there is a path of tree edges from v down to u, which is in particular a directed path from v to u in G. The arc

(u, v) closes it into a directed cycle. Hence G is not acyclic. (If $u = v$ the "path" is empty and the arc is a self-loop, which is a directed cycle of length 1.)

(⇐, in contrapositive form: a cycle implies a back edge.) Suppose G contains a directed cycle c. Among the vertices of c, let v be the **first one discovered** by the DFS, and let u be the vertex immediately before v on the cycle, so (u, v) is an arc of c.

At the instant $d[v]$ when v is discovered, every other vertex of c is still white, because v was the first of them to be discovered. So the stretch of the cycle running from v forward all the way round to u is a path of white vertices from v to u. By the white-path theorem (Theorem 10.8), u therefore becomes a **descendant** of v in the depth-first forest.

Now consider the arc (u, v), examined during DFS-Visit(u). Since u is a descendant of v, Corollary 10.7 gives $d[v] < d[u] < f[u] < f[v]$ — so at the moment (u, v) is examined, the clock reads somewhere inside $[d[u], f[u]] \subset [d[v], f[v]]$, which means v is still **gray**. An arc into a gray vertex is a back edge. ■

A compact summary of the second half is worth memorizing: *upon discovering v the whole cycle from v to u is white; we must visit all nodes reachable on this white path before returning from DFS-Visit(v), i.e., vertex u becomes a descendant of v; thus (u, v) is a back edge.*

Theorem 10.10 is exactly what **Listing 10.7**'s directed test implements, and it also explains the warning in §10.9.2: the theorem says *back* edges, and only the gray test finds back edges.

10.9.6 Topological sort

Problem. Given a DAG, list all its vertices in an order such that for every arc (u, v), u comes before v.

> **DEFINITION.** A **topological sort** of a DAG $G = (V, E)$ is a linear ordering of V such that for every arc $(u, v) \in E$, u appears before v in the ordering.

Picture the vertices laid out on a horizontal line; a topological sort is an arrangement in which **every arrow points to the right**. **Figure 10.30** is exactly that picture.

Why it must be a DAG: if there were a directed cycle $v_1 \to v_2 \to \cdots \to v_k \to v_1$, then v_1 would have to come before v_2, which comes before v_3, ..., which comes before v_1 — so v_1 comes before itself. Impossible. Conversely, we are about to prove that every DAG *has* a topological sort, so "is a DAG" and "has a topological sort" are the same condition.

The algorithm is **Pseudocode 10.3**, and it takes three lines:

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
```

So the topological order is the vertices **in decreasing order of finishing time** — and step 2 achieves that without any sorting, because inserting each vertex at the front as it finishes automatically reverses the finishing order.

The example. We use the classic dressing problem: the garments a person puts on in the morning, with an arrow from x to y meaning *one must be done with x before one can do y*. The

intuition in one line: *a task can be scheduled only when all of its subtasks have been scheduled.* **Figure 10.29** is the DAG with the DFS timestamps on it, and **Figure 10.30** is the resulting order.

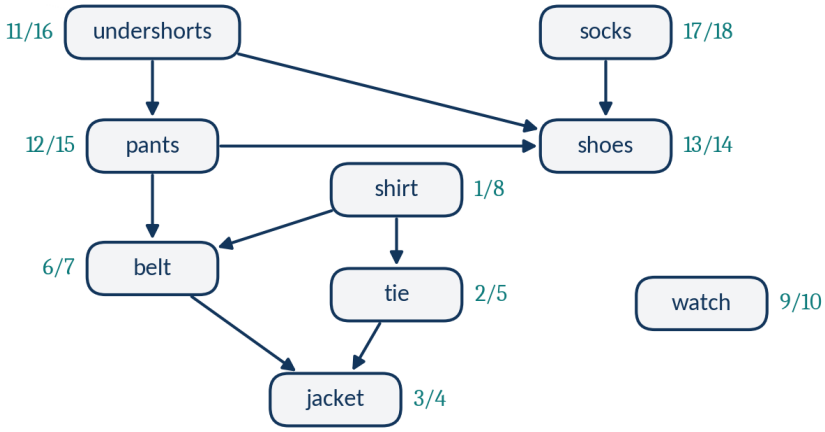


Figure 10.29 The dressing DAG, timestamped by a DFS. Nine garments; an arrow from x to y means x must be put on before y. The stamps are shirt 1/8, tie 2/5, jacket 3/4, belt 6/7, watch 9/10, undershorts 11/16, pants 12/15, shoes 13/14, socks 17/18 — eighteen stamps for nine vertices, as $2|V|$ requires. The watch has no arrows at all: it may be put on at any time.

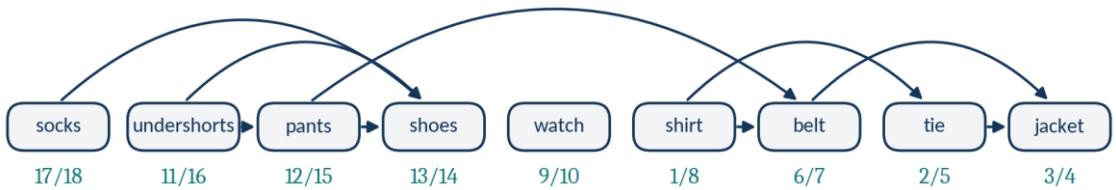


Figure 10.30 The topological order: the same nine vertices laid out left to right in decreasing order of finishing time — socks 18, undershorts 16, pants 15, shoes 14, watch 10, shirt 8, belt 7, tie 5, jacket 4 — and now **every arrow points right**. That is what a topological sort is.

Correctness. The proof, which we shall set out in full, rests on a single claim.

> **THEOREM 10.11.** *If G is a DAG, then for every arc $(u, v) \in E$ we have $f[v] < f[u]$. Consequently, listing the vertices in decreasing order of f is a topological sort.*

Proof. Consider the moment the arc (u, v) is examined. At that moment u is gray, since we are inside DFS-Visit(u). There are exactly three possibilities for v's color:

- **v is gray.** Then (u, v) is a back edge, and by Theorem 10.10 G has a directed cycle — contradicting the assumption that G is a DAG. So this case cannot happen.
- **v is white.** Then DFS-Visit(v) is called immediately, from inside DFS-Visit(u). So v becomes a descendant of u, and by the parenthesis theorem v's interval nests inside u's: $f[v] < f[u]$.
- **v is black.** Then v has already finished, and u has not — it is still gray. So $f[v]$ is already stamped and $f[u]$ is not: $f[v] < f[u]$.

In the two possible cases, $f[v] < f[u]$. So for every arc, the head finishes before the tail. Now list the vertices in decreasing order of f. For any arc (u, v), $f[u] > f[v]$ puts u earlier in the list than v. That is precisely the definition of a topological sort. ■

Cost. The accounting is short: *depth-first search takes $O(V + E)$ time (by* Equation (10.11)*); inserting each of the $|V|$ vertices onto the front of the linked list costs $O(1)$ per insertion. Thus the total running time is $O(V + E)$.* Note that no *sort* is performed, despite the name — inserting at the front as each vertex finishes gives the decreasing-f order for free. Had we instead collected the pairs $(v, f[v])$ and sorted them, the cost would have been $O(V \log V + E)$, which is worse for no benefit.

Not unique. A DAG usually has many topological sorts. In **Figure 10.30** the watch sits between shoes and shirt, but it could go anywhere: it has no arrows. Which order you get depends on the order the outer DFS loop tries vertices and the order of each adjacency list. Any of them is a correct answer; if a program must produce a specific one — say, the lexicographically smallest — a different algorithm is needed (see Exercise 10.14).

Listing 10.9 runs the dressing example and then *verifies* its own output, by checking that every arc really does point rightwards.

Listing 10.9 Topological sort of the dressing DAG by depth-first search, followed by a verification pass that checks every arc points forward in the produced order.

```
// Topological sort by DFS: as each vertex finishes, push it on the front of a
// list. The graph is the dressing example of Section 10.9.6.
#include <iostream>
#include <vector>
#include <list>
#include <string>
#include <iomanip>
using namespace std;

enum Color { WHITE, GRAY, BLACK };

struct TopoSort {
    vector<vector<int>> adj;
    vector<string> name;
    vector<Color> color;
    vector<int> d, f;
    int time = 0;
    list<int> order;                      // built by pushing onto the front
    bool cyclic = false;

    TopoSort(vector<vector<int>> g, vector<string> nm)
        : adj(move(g)), name(move(nm)), color(adj.size(), WHITE),
          d(adj.size(),0), f(adj.size(),0) {}

    void run() { for (size_t u = 0; u < adj.size(); ++u) if (color[u]==WHITE)
visit((int)u); }

    void visit(int u) {
        color[u] = GRAY; d[u] = ++time;
        for (int v : adj[u]) {
            if (color[v] == GRAY) cyclic = true;      // a back edge: not a DAG
            if (color[v] == WHITE) visit(v);
        }
        color[u] = BLACK; f[u] = ++time;
        order.push_front(u);                           // O(1) per vertex
    }
};
```

Listing 10.9 (Continued) *Topological sort of the dressing DAG by depth-first search, followed by a verification pass that checks every arc points forward in the produced order.*

```
int main() {
    // The main loop scans vertices in this index order, which is what fixes the
    // timestamps below.
    vector<string> name = {"shirt","tie","jacket","belt","watch",
                           "undershorts","pants","shoes","socks"};
    vector<vector<int>> adj(9);
    adj[0] = {1,3};        // shirt       -> tie, belt
    adj[1] = {2};          // tie         -> jacket
    adj[3] = {2};          // belt        -> jacket
    adj[5] = {6,7};        // undershorts -> pants, shoes
    adj[6] = {7,3};        // pants       -> shoes, belt
    adj[8] = {7};          // socks       -> shoes
    // watch (4) and jacket (2) and shoes (7) have no outgoing arcs

    TopoSort t(adj, name);
    t.run();

    cout << "discovery and finishing times\n";
    for (size_t v = 0; v < adj.size(); ++v)
        cout << "  " << setw(12) << left << t.name[v] << " " << t.d[v] << "/" << t.f[v] <<
"\n";

    cout << "\nthe graph " << (t.cyclic ? "has a directed cycle; no topological order
exists"
                                         : "is a DAG") << "\n";

    cout << "\ntopological order (decreasing finishing time)\n  ";
    for (int v : t.order) cout << t.name[v] << " ";
    cout << "\n";

    cout << "\nverification: every arc goes left to right\n";
    vector<int> pos(adj.size());
    int i = 0;
    for (int v : t.order) pos[v] = i++;
    bool allOK = true;
    for (size_t u = 0; u < adj.size(); ++u)
        for (int v : adj[u]) {
            bool ok = pos[u] < pos[v];
            allOK = allOK && ok;
            cout << "  " << setw(12) << left << t.name[u] << " -> "
                 << setw(7) << left << t.name[v]
                 << "  position " << pos[u] << " -> " << pos[v]
                 << (ok ? "   ok" : "   VIOLATION") << "\n";
        }
    cout << (allOK ? "\nall arcs respected.\n" : "\nthe order is wrong.\n");
    return 0;
}
```

Its output:

```
discovery and finishing times
  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

the graph is a DAG

topological order (decreasing finishing time)
  socks undershorts pants shoes watch shirt belt tie jacket

verification: every arc goes left to right
  shirt        -> tie      position 5 -> 7   ok
  shirt        -> belt     position 5 -> 6   ok
  tie          -> jacket   position 7 -> 8   ok
  belt         -> jacket   position 6 -> 8   ok
  undershorts  -> pants    position 1 -> 2   ok
  undershorts  -> shoes    position 1 -> 3   ok
  pants        -> shoes    position 2 -> 3   ok
  pants        -> belt     position 2 -> 6   ok
  socks        -> shoes    position 0 -> 3   ok

all arcs respected.
```

Every timestamp is exactly the one in **Figure 10.29**, and the order printed is exactly the one in **Figure 10.30**: *socks, undershorts, pants, shoes, watch, shirt, belt, tie, jacket*. All nine arcs point rightwards. The program agrees with the figures line for line.

Looking ahead. Chapter 11 returns to topological sort and does two things with it that this chapter cannot. First, it gives the *other* algorithm — repeatedly remove a vertex of in-degree zero, known as Kahn's algorithm — which is also $O(V + E)$ and has the advantage of reporting *which* vertices are involved in a cycle when the graph is not a DAG. Second, it uses the topological order as the scaffolding for shortest paths in a DAG, which can be computed in linear time and with negative weights — something Dijkstra cannot do.

10.9.7 So: breadth-first or depth-first?

Table 10.12 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

The practical rule is short. **If the question mentions distance, hops, nearest or level, use BFS. If the question is about structure — order, cycles, dependency, reachability without a metric — use DFS.** And if neither, either will do; use whichever you can write correctly on the first attempt.

10.9.8 Measured

A claim as strong as $O(V+E)$ deserves to be checked against a clock rather than only against a proof. **Listing 10.10** builds a random connected graph with $|E| = 4|V|$, runs BFS over an adjacency list and the same BFS over an adjacency matrix, and reports the time per unit of $(V+E)$ and per unit of V^2. If the theory is right, the first of those should stay roughly constant as the graph grows, and so should the second.

Listing 10.10 Timing BFS over an adjacency list against BFS over an adjacency matrix on the same graphs. A spanning tree is built first, so the graph is certainly connected and both searches must reach all $|V|$ vertices — which the program checks before reporting any time.

```
// Measuring the claim. BFS over an adjacency list should cost Theta(V+E);
// the same BFS over an adjacency matrix must pay Theta(V) per dequeued vertex
// whether the edges are there or not, so Theta(V^2).
// The test graph is a random connected graph with |E| = 4|V|.
#include <iostream>
#include <vector>
#include <queue>
#include <random>
#include <chrono>
#include <iomanip>
using namespace std;
using namespace std::chrono;

static long long bfsList(const vector<vector<int>>& adj, int s) {
    int n = (int)adj.size();
    vector<int> d(n, -1);
    queue<int> Q; Q.push(s); d[s] = 0;
    long long visited = 0;
    while (!Q.empty()) {
        int u = Q.front(); Q.pop(); ++visited;
        for (int v : adj[u]) if (d[v] < 0) { d[v] = d[u]+1; Q.push(v); }
    }
    return visited;
}

static long long bfsMatrix(const vector<vector<char>>& M, int s) {
    int n = (int)M.size();
    vector<int> d(n, -1);
    queue<int> Q; Q.push(s); d[s] = 0;
    long long visited = 0;
    while (!Q.empty()) {
        int u = Q.front(); Q.pop(); ++visited;
        for (int v = 0; v < n; ++v)                 // every v, edge or no edge
            if (M[u][v] && d[v] < 0) { d[v] = d[u]+1; Q.push(v); }
    }
    return visited;
}
```

Listing 10.10 (Continued) *Timing BFS over an adjacency list against BFS over an adjacency matrix on the same graphs. A spanning tree is built first, so the graph is certainly connected and both searches must reach all |V| vertices — which the program checks before reporting any time.*

```
int main() {
    cout << "     |V|      |E|   list (ms)  matrix (ms)   list/(V+E) ns   matrix/V^2 ns\n";
    mt19937 rng(20260919);
    for (int n : {1000, 2000, 4000, 8000}) {
        int m = 4 * n;
        vector<vector<int>> adj(n);
        vector<vector<char>> M(n, vector<char>(n, 0));
        auto add = [&](int a, int b) {
            if (a == b || M[a][b]) return false;
            M[a][b] = M[b][a] = 1; adj[a].push_back(b); adj[b].push_back(a); return true;
        };
        for (int v = 1; v < n; ++v) add(v, (int)(rng() % v));   // a spanning tree first
        int placed = n - 1;
        while (placed < m) if (add((int)(rng()%n), (int)(rng()%n))) ++placed;

        auto t0 = steady_clock::now();
        long long a = bfsList(adj, 0);
        auto t1 = steady_clock::now();
        long long b = bfsMatrix(M, 0);
        auto t2 = steady_clock::now();
        if (a != n || b != n) { cout << "graph not connected\n"; return 1; }

        double msL = duration<double, milli>(t1 - t0).count();
        double msM = duration<double, milli>(t2 - t1).count();
        double perVE = msL * 1e6 / (n + m);
        double perV2 = msM * 1e6 / ((double)n * n);
        cout << setw(8) << n << setw(9) << m
             << setw(12) << fixed << setprecision(3) << msL
             << setw(13) << msM
             << setw(17) << setprecision(2) << perVE
             << setw(16) << perV2 << "\n";
    }
    return 0;
}
```

Its output on the machine used to prepare this chapter:

```
     |V|      |E|   list (ms)  matrix (ms)   list/(V+E) ns   matrix/V^2 ns
    1000     4000       0.037        0.778             7.38            0.78
    2000     8000       0.072        3.105             7.25            0.78
    4000    16000       0.139       12.263             6.94            0.77
    8000    32000       0.353       47.902             8.83            0.75
```

Table 10.13 The measurements of **Listing 10.10**, with the growth ratios added. Absolute times depend on the machine and vary by a few tens of percent from run to run, so it is the *ratios* that carry the claim: doubling the graph roughly **doubles** the adjacency-list time and very nearly **quadruples** the matrix time — the signatures of $\Theta(V + E)$ and $\Theta(V^2)$ respectively. The last two columns, the time divided by the predicted cost, stay within a small factor across an eight-fold range of sizes, which is what "linear" and "quadratic" mean in practice.

$\|V\|$	$\|E\|$	List, ms	Matrix, ms	List time ÷ previous	Matrix time ÷ previous	Ns per $(V+E)$	Ns per V^2
1 000	4 000	0.037	0.778	—	—	7.38	0.78
2 000	8 000	0.072	3.105	1.95	3.99	7.25	0.78
4 000	16 000	0.139	12.263	1.93	3.95	6.94	0.77
8 000	32 000	0.353	47.902	2.54	3.91	8.83	0.75

Read the table carefully, because it says two different things.

The list version is linear. Its time roughly doubles when the graph doubles — the ratios are 1.95, 1.93, 2.54 — and its time per unit of $(V + E)$ stays between 6.9 and 8.9 nanoseconds across an eightfold increase in size. It does not stay *exactly* constant, and it would be dishonest to claim it did: the last row drifts upward because at 8 000 vertices the adjacency lists no longer fit in the processor's fast cache and some neighbor lookups must reach into slower memory. Asymptotic analysis deliberately ignores such effects, and this is what "deliberately ignores" looks like on a real machine — a drift of about a quarter, against a factor of eight in input size.

The matrix version is quadratic. Its times are 0.778, 3.105, 12.263, 47.902 — each very nearly four times the one before, the ratios being 3.99, 3.95 and 3.91, exactly as V^2 requires when V doubles. And its time per V^2 is flat to within four percent, between 0.75 and 0.78 nanoseconds.

The gap widens without limit. At 1 000 vertices the matrix version is about 21 times slower. At 8 000 it is about 136 times slower. Why? The matrix version costs about 0.77 ns per cell, V^2 cells in all; the list version costs about 7.5 ns per entry, $V + E = 5V$ entries in all. The ratio is therefore about $0.77\,V^2/(7.5 \times 5V) \approx V/49$ — it grows in proportion to V. That formula gives about 160 at $V = 8\,000$, close to the 136 measured, and at a million vertices it predicts a matrix search about twenty thousand times slower — if the matrix could be allocated at all, which by **Table 10.4** it could not. This is the concrete content of the warning in §10.7.7.

10.10 Profitina in Practice

Profitina is an **AI-visibility (GEO) platform**: it measures how ready a brand or a website is to be *cited by AI answer engines*, from real signals — how extractable the content is, whether structured data is present and well formed, what technical authority the site carries — and then helps the owner fix what it finds. It is live in production at `app.profitina.com`. Generative Engine Optimization is to answer engines what search-engine optimization was to lists of links.

Where does this chapter appear in that product? In the very first thing the scanner does.

A website is a directed graph, and the scan is a graph search. The vertices are pages — more precisely, canonical URLs. The edges are hyperlinks, and they are directed: a link from the home page to the pricing page does not give you a link back. To score a site, the scanner must visit the pages reachable from a starting URL, visit each one exactly once, and stop. That is breadth-first search, with the three colors of §10.6 and not one idea more:

- **white** — a URL we have seen mentioned but not yet fetched: it sits in the frontier queue;
- **gray** — fetched, and its links now being extracted;
- **black** — fetched, links extracted, signals scored.

Forget the colors and the crawler revisits pages forever, because real websites are full of cycles: every page links to the home page, which links back. That is not a hypothetical bug; it is the first bug every crawler has.

Why breadth-first rather than depth-first. Three reasons, and they are the reasons this chapter spent so long on **Table 10.12**.

First, **the pages that matter are near the home page.** A site's important pages — home, product, pricing, about, contact — are typically at distance 0, 1 or 2 from the root. BFS visits them first, by Theorem 10.4, so a scan that is cut short after a budget of pages has still seen the pages that carry the most weight. A depth-first crawl might spend its entire budget in one deep branch of a blog archive from 2019.

Second, **"click depth" is a real GEO signal, and BFS computes it for free.** The $d[v]$ that BFS produces is exactly the number of clicks from the home page to page v — the fewest-hops distance of §10.9.4. A page buried six clicks deep is harder for a crawler, human or machine, to reach, and that is worth telling the site owner. The array is already there; no extra work is needed.

Third, **BFS is polite.** A breadth-first frontier naturally spreads requests across the site rather than hammering one directory, and it interleaves neatly with a per-host rate limit: you take the head of the queue, wait your interval, take the next.

Where the other algorithms of this chapter appear.

Connected components (§10.9.1) identify **orphan clusters** — groups of pages that link to each other but that nothing reachable from the home page links to. They are invisible to a crawler entering at the root, which means they are invisible to an answer engine, which is a finding worth reporting.

Cycle detection (§10.9.2) is not run on the page graph, where cycles are normal and harmless, but on the **signal-dependency graph**: some signals are computed from others, and the pipeline that evaluates them is a DAG. If a configuration change ever made that graph cyclic, the pipeline would either loop or deadlock. A cycle check at start-up, costing $O(V + E)$ on a graph of a few dozen vertices, turns a production hang into an error message.

Topological sort (§10.9.6) then gives the **order in which the signals are evaluated**, so that nothing is computed before its inputs. This is the same computation as **Listing 10.9**, on a different DAG.

The representation choice of §10.5 is decided by **Table 10.6** without hesitation: a site with 50 000 pages and 400 000 links is emphatically sparse, so it is stored as adjacency lists. An adjacency matrix would need 2.5×10^9 entries for a graph whose adjacency lists need under a million, and at the per-entry costs measured in **Table 10.13** the crawl would also be several hundred times slower — about $0.77 \times 2.5 \times 10^9$ ns against $7.5 \times 4.5 \times 10^5$ ns, a factor of about 570.

Two constraints from outside the algorithm are worth naming, because in a real product they matter as much as the asymptotics. **UU PDP 27/2022** governs personal data, so a page graph is stored as URLs and link structure and nothing else — no page contents that could contain personal data are retained beyond what the scoring needs. And the product **advises; the user decides**: a click-depth finding is presented as a measured fact with an explanation, never as a guarantee about how any particular answer engine will behave.

10.10.1 The worked case study: Profitina Laundry

The runnable case study in this book is always **Profitina Laundry**, a small teaching business, not the real product. It is a laundry with one outlet in Keputih, near the ITS campus in Surabaya, a motorbike courier, and an ambition to promise same-day pickup.

Two questions arrive on the same morning, and both are graph questions.

Question 1. "Which neighborhoods can we promise a two-hop pickup to?" The courier can ride from one *kelurahan* to an adjacent one; the outlet is in Keputih. The vertices are the neighborhoods served; there is an edge between two neighborhoods if the courier can ride directly between them. Every ride costs roughly the same time, so this is an **unweighted** graph and the question is fewest hops: BFS from Keputih, and read $d[v]$. (If the rides had different

durations we would be in Chapter 13, and this would be the wrong algorithm — the warning of §10.4.14 applies to laundry as much as to anything else.)

The BFS answers a second question for free. Any neighborhood with $d[v] = \infty$ is **not reachable at all** — it is in a different connected component, and no courier route exists. That is not a scheduling problem; it is a second-outlet problem, and it is worth knowing before a promise is printed on a banner.

Question 2. "In what order must a bag of laundry be worked?" Intake, weigh, sort, pretreat, wash, dry, iron, fold, pack, invoice, notify. Some of these must precede others — you cannot dry before you wash — and some are independent: the invoice can be raised any time after weighing. The vertices are stations; an arc from x to y means x must finish before y may start. That is a DAG, and a legal work order is a topological sort. And if someone writes down a set of rules that is *not* a DAG, the program must say so and name the cycle, rather than produce a plausible-looking order that cannot be followed.

Listing 10.11 answers both.

Listing 10.11 The Profitina Laundry case study: breadth-first search on the pickup network of neighborhoods, and topological sort on the wash-process DAG, together with what happens when the process specification is made cyclic.

```
// Profitina Laundry: two graph problems from one working day.
//   Part 1  the pickup network of an outlet, as an undirected graph of
//           neighborhoods; BFS answers "how many hops away, and by which
//           route", and the components tell us which zones no courier reaches.
//   Part 2  the wash-process graph, as a DAG; a topological sort is a legal
//           order of work, and a back edge is a specification bug.
#include <iostream>
#include <vector>
#include <string>
#include <queue>
#include <list>
#include <algorithm>
#include <iomanip>
#include <climits>
using namespace std;

// ---------------------------------------------------------- Part 1: BFS
struct Zones {
    vector<string> name;
    vector<vector<int>> adj;
    int edges = 0;

    int add(const string& s) { name.push_back(s); adj.emplace_back(); return
(int)name.size()-1; }
    void road(int a, int b) { adj[a].push_back(b); adj[b].push_back(a); ++edges; }
    int  n() const { return (int)name.size(); }
};

struct BfsResult { vector<int> d, pi; long long edgesScanned = 0; };

BfsResult bfs(const Zones& Z, int s) {
    BfsResult R;
    R.d.assign(Z.n(), INT_MAX);
    R.pi.assign(Z.n(), -1);
    R.d[s] = 0;
    queue<int> Q; Q.push(s);
    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        for (int v : Z.adj[u]) {
            ++R.edgesScanned;                        // each directed slot once
            if (R.d[v] == INT_MAX) { R.d[v] = R.d[u] + 1; R.pi[v] = u; Q.push(v); }
        }
    }
    return R;
}
```

Listing 10.11 (Continued) *The Profitina Laundry case study: breadth-first search on the pickup network of neighborhoods, and topological sort on the wash-process DAG, together with what happens when the process specification is made cyclic.*

```
// ---------------------------------------------------------- Part 2: DFS
enum Color { WHITE, GRAY, BLACK };

struct Process {
    vector<string> name;
    vector<vector<int>> adj;
    int add(const string& s) { name.push_back(s); adj.emplace_back(); return
(int)name.size()-1; }
    void before(int a, int b) { adj[a].push_back(b); }   // a must finish before b
    int  n() const { return (int)name.size(); }
};

struct Topo {
    const Process& P;
    vector<Color> color;
    list<int> order;
    bool cyclic = false;
    string cycle;                                   // the cycle, spelled out
    vector<int> grayStack;                          // the chain of gray vertices
    explicit Topo(const Process& p) : P(p), color(p.n(), WHITE) {}
    void run() { for (int u = 0; u < P.n(); ++u) if (color[u]==WHITE) visit(u); }
    void visit(int u) {
        color[u] = GRAY;
        grayStack.push_back(u);
        for (int v : P.adj[u]) {
            if (color[v] == GRAY && !cyclic) {      // a back edge closes a cycle
                cyclic = true;
                size_t i = 0;
                while (grayStack[i] != v) ++i;       // v is somewhere below u
                for (; i < grayStack.size(); ++i) cycle += P.name[grayStack[i]] + " -> ";
                cycle += P.name[v];
            }
            if (color[v] == WHITE) visit(v);
        }
        color[u] = BLACK;
        grayStack.pop_back();
        order.push_front(u);
    }
};
```

Listing 10.11 (Continued) *The Profitina Laundry case study: breadth-first search on the pickup network of neighborhoods, and topological sort on the wash-process DAG, together with what happens when the process specification is made cyclic.*

```
int main() {
    cout << "================ Part 1: the pickup network ================\n";
    Zones Z;
    int keputih   = Z.add("Keputih");
    int gebang    = Z.add("Gebang Putih");
    int klampis   = Z.add("Klampis Ngasem");
    int semolo    = Z.add("Semolowaru");
    int sukolilo  = Z.add("Sukolilo");
    int medokan   = Z.add("Medokan Semampir");
    int manyar    = Z.add("Manyar Sabrangan");
    int nginden   = Z.add("Nginden Jangkungan");
    int mulyorejo = Z.add("Mulyorejo");
    int kalijudan = Z.add("Kalijudan");
    Z.add("Dukuh Setro");   // across the river: no road reaches it, so it needs no name
here

    Z.road(keputih, gebang);
    Z.road(keputih, sukolilo);
    Z.road(gebang, klampis);
    Z.road(gebang, mulyorejo);
    Z.road(klampis, semolo);
    Z.road(klampis, manyar);
    Z.road(semolo, nginden);
    Z.road(semolo, medokan);
    Z.road(sukolilo, medokan);
    Z.road(manyar, nginden);
    Z.road(mulyorejo, kalijudan);
    // Dukuh Setro is deliberately left with no road

    cout << "|V| = " << Z.n() << " zones, |E| = " << Z.edges << " roads\n";
    BfsResult R = bfs(Z, keputih);
    cout << "\nBFS from the outlet in " << Z.name[keputih] << "\n";
    cout << "  " << setw(20) << left << "zone" << setw(6) << "hops" << "route\n";
    for (int v = 0; v < Z.n(); ++v) {
        cout << "  " << setw(20) << left << Z.name[v];
        if (R.d[v] == INT_MAX) { cout << setw(6) << "-" << "unreachable by road\n";
continue; }
        cout << setw(6) << R.d[v];
        vector<int> p;
        for (int x = v; x != -1; x = R.pi[x]) p.push_back(x);
        reverse(p.begin(), p.end());
        for (size_t i = 0; i < p.size(); ++i) cout << (i ? " > " : "") << Z.name[p[i]];
        cout << "\n";
    }
    cout << "\n  edge slots scanned = " << R.edgesScanned
         << ", and 2|E| = " << 2*Z.edges << "\n";
```

Listing 10.11 (Continued) *The Profitina Laundry case study: breadth-first search on the pickup network of neighborhoods, and topological sort on the wash-process DAG, together with what happens when the process specification is made cyclic.*

```
    int far = 0;
    for (int v = 0; v < Z.n(); ++v)
        if (R.d[v] != INT_MAX && R.d[v] > R.d[far]) far = v;
    cout << "  furthest reachable zone: " << Z.name[far] << " at " << R.d[far] << "
hops\n";
    cout << "  a 2-hop promise covers:";
    for (int v = 0; v < Z.n(); ++v)
        if (R.d[v] != INT_MAX && R.d[v] <= 2) cout << " " << Z.name[v] << ",";
    cout << " and nothing else\n";

    cout << "\n================ Part 2: the wash process ================\n";
    Process P;
    int intake  = P.add("intake");
    int weigh   = P.add("weigh");
    int sort_   = P.add("sort");
    int pretreat= P.add("pretreat");
    int wash    = P.add("wash");
    int dry     = P.add("dry");
    int iron    = P.add("iron");
    int fold    = P.add("fold");
    int pack    = P.add("pack");
    int invoice = P.add("invoice");
    int notify  = P.add("notify");

    P.before(intake, weigh);
    P.before(weigh, sort_);
    P.before(weigh, invoice);
    P.before(sort_, pretreat);
    P.before(sort_, wash);
    P.before(pretreat, wash);
    P.before(wash, dry);
    P.before(dry, iron);
    P.before(dry, fold);
    P.before(iron, fold);
    P.before(fold, pack);
    P.before(invoice, pack);
    P.before(pack, notify);

    Topo T(P);
    T.run();
    cout << "is a DAG: " << (T.cyclic ? "no" : "yes") << "\n";
    cout << "one legal order of work:\n  ";
    for (int v : T.order) cout << P.name[v] << " ";
    cout << "\n";
```

Listing 10.11 (Continued) *The Profitina Laundry case study: breadth-first search on the pickup network of neighborhoods, and topological sort on the wash-process DAG, together with what happens when the process specification is made cyclic.*

```
    // Now break the specification: insist that the invoice be issued only
    // after the customer has been notified.
    Process Bad = P;
    Bad.before(notify, invoice);
    Topo T2(Bad);
    T2.run();
    cout << "\nafter adding 'notify -> invoice':\n";
    cout << "  is a DAG: " << (T2.cyclic ? "no" : "yes") << "\n";
    if (T2.cyclic) cout << "  the cycle is: " << T2.cycle
                        << "\n  no order of work can satisfy this\n";
    return 0;
}
```

Its output:

```
================ Part 1: the pickup network ================
|V| = 11 zones, |E| = 11 roads

BFS from the outlet in Keputih
  zone                hops  route
  Keputih             0     Keputih
  Gebang Putih        1     Keputih > Gebang Putih
  Klampis Ngasem      2     Keputih > Gebang Putih > Klampis Ngasem
  Semolowaru          3     Keputih > Gebang Putih > Klampis Ngasem > Semolowaru
  Sukolilo            1     Keputih > Sukolilo
  Medokan Semampir    2     Keputih > Sukolilo > Medokan Semampir
  Manyar Sabrangan    3     Keputih > Gebang Putih > Klampis Ngasem > Manyar Sabrangan
  Nginden Jangkungan  4     Keputih > Gebang Putih > Klampis Ngasem > Semolowaru > Nginden
Jangkungan
  Mulyorejo           2     Keputih > Gebang Putih > Mulyorejo
  Kalijudan           3     Keputih > Gebang Putih > Mulyorejo > Kalijudan
  Dukuh Setro         -     unreachable by road

  edge slots scanned = 22, and 2|E| = 22
  furthest reachable zone: Nginden Jangkungan at 4 hops
  a 2-hop promise covers: Keputih, Gebang Putih, Klampis Ngasem, Sukolilo, Medokan
Semampir, Mulyorejo, and nothing else

================ Part 2: the wash process ================
is a DAG: yes
one legal order of work:
  intake weigh invoice sort pretreat wash dry iron fold pack notify

after adding 'notify -> invoice':
  is a DAG: no
  the cycle is: pack -> notify -> invoice -> pack
  no order of work can satisfy this
```

Five things in that output are worth pointing at.

The answer to question 1 is a business decision made computable. Six of the eleven zones are within two hops; the two-hop promise can honestly be printed for those six and no others. Nginden Jangkungan is four hops away and should be quoted differently, or served from a second outlet.

Dukuh Setro is reported as unreachable, not as far away. Its d value stayed at ∞, which is the "not known to be reachable" of §10.7.2 doing its job. Reporting it as a large number would have been a lie; reporting it as unreachable is a finding. This is the practical reason to initialize distances to ∞ rather than to -1 or 0.

The edge-slot count is exactly $2|E|$. The program counted 22 examinations for 11 undirected edges. That is Consequence 2 of the handshake lemma (§10.4.3) measured rather than proved, and it is the concrete content of the $O(E)$ term in §10.7.7: the search looked at each edge from each end, once, and never again.

The routes come out of π, not out of a second search. Every route printed was reconstructed by walking predecessors back to the outlet and reversing — the operation of §10.7.5. One BFS produced eleven routes.

The cycle is named, not merely detected. When the rule *notify before invoice* is added, the program reports `pack -> notify -> invoice -> pack`. That is the chain of gray vertices at the moment the back edge was found, which is precisely why the gray stack was kept: an error message that says "there is a cycle" sends someone hunting; an error message that names the

three steps involved ends the conversation. Theorem 10.10 is what guarantees that finding a back edge is the same thing as finding a cycle.

10.11 Common Mistakes, and How to Avoid Them

1. Forgetting to mark a vertex as seen. The program runs forever, or until memory is exhausted. This is the mistake, and every one of the others is a refinement of it. The cure is discipline: the line that discovers a vertex and the line that marks it must be written *at the same time*, before you write anything else in the loop.

2. Marking on dequeue instead of on enqueue. In BFS, mark v gray at the moment you push it (line 13 of **Figure 10.15**), not when you later pop it. If you mark on pop, a vertex with three unmarked neighbors pointing at it enters the queue three times; the program still terminates and still computes correct distances, but the queue can grow to $\Theta(E)$ instead of $\Theta(V)$ and the running time degrades. On a dense graph this is the difference between a program that works and one that runs out of memory.

3. Using BFS on a weighted graph. Described at length in §10.4.14 and §10.9.4, and repeated here because it is the most expensive error in the chapter: BFS minimizes the number of edges, not the total weight. The symptom is an answer that is *plausible but wrong*, which is far worse than a crash. Chapter 13 is the fix.

4. Using `visited` where you need three colors. For directed cycle detection the test is "is v **gray**", not "have I seen v" (§10.9.2). A two-state marker reports forward and cross edges as cycles. The graph $u \to v, u \to w, v \to w$ is acyclic and a `visited`-based test says otherwise.

5. Forgetting the parent in undirected cycle detection. Conversely, in an *undirected* graph every tree edge looks like a back edge from the child's point of view, because the parent is gray and is in the child's adjacency list. You must skip the edge you arrived on — and skip it exactly once, so that genuine parallel edges are still caught (**Listing 10.7**).

6. Dequeueing with `vector::erase(begin())`. It is a $\Theta(V)$ operation, and doing it once per vertex makes the whole search $\Theta(V^2)$. Use `std::queue`, `std::deque`, or an array with a head index. The same warning applies to `list::size()` in old implementations and to any "remove from the front" that shifts elements.

7. Recursing on a deep graph. A recursive DFS on a path of a million vertices overflows the call stack and the program dies with no useful message. If the graph may be deep, use the iterative form of **Listing 10.3** — and remember that it needs the resume index, not just a stack of vertices.

8. Resuming an adjacency list from the beginning. The specific bug that catches people converting a recursive DFS to an iterative one: on returning to a vertex, they rescan $\mathrm{Adj}[u]$ from index 0. The search then loops or takes quadratic time. The `Frame { int u; size_t next; }` of **Listing 10.3** exists precisely to prevent this.

9. Building an adjacency list with only one direction, in an undirected graph. `adj[u].push_back(v)` without `adj[v].push_back(u)`. The search then works from one end and silently fails from the other. Write a single `addEdge` helper that does both, and never push directly.

10. Allocating a $V \times V$ matrix for a sparse graph. $|V| = 10^6$ needs 10^{12} entries. The allocation fails, or worse, succeeds on a machine with overcommitted memory and then thrashes. **Table 10.4** is the table to consult before writing `vector<vector<char>> M(n, vector<char>(n))`.

11. Assuming one search covers the whole graph. BFS from s reaches only s's component. If the task is "every vertex", you need the outer loop over all vertices that DFS has built in and BFS does not (**Listing 10.6**).

12. Reading $d[v] = \infty$ as "very far". It means *unreachable*. Printing it, comparing it, or adding one to it are all bugs; in C++, `INT_MAX + 1` is signed overflow and therefore undefined behavior. Test for the sentinel explicitly, as **Listing 10.11** does.

13. Expecting a unique answer. The distances a BFS computes are unique; the tree, the π array, the order of the queue, the DFS timestamps and the topological order are not — they all depend on the order of the adjacency lists (§10.7.4, §10.9.6). Two correct programs can print different trees. Only compare what is actually determined.

14. Confusing d in BFS with d in DFS. In BFS, $d[v]$ is a *distance*. In DFS, $d[u]$ is a *discovery time*. They are unrelated quantities that share a letter because the standard reference [1] uses it for both. When you write both algorithms in one program, rename one of them.

15. Claiming $O(V + E)$ while using a matrix. The bound is a statement about adjacency lists (§10.7.7). **Table 10.13** is what the same code costs over a matrix.

10.12 Chapter Summary

A **graph** $G = (V, E)$ is a set of vertices and a set of edges, each edge a pair of vertices; the pairs are ordered in a directed graph and unordered in an undirected one (**Figure 10.1**). The drawing is not the graph.

The **vocabulary** is collected in one table, **Table 10.3**. The one result in it that is used everywhere is the **handshake lemma** Equation (10.1), $\sum_v \deg(v) = 2|E|$, proved in §10.4.3 by counting edge-ends two ways; its programmer's form is that all the adjacency lists together hold $2|E|$ entries, and that fact is the source of the E in every running time in the chapter.

There are **three representations**. The *edge list* is best only at listing all edges. The *adjacency list* uses $\Theta(V + E)$ space and answers "list the neighbors of v" optimally, which is the only question a search asks; it is the default. The *adjacency matrix* uses $\Theta(V^2)$ space and answers "is (u, v) an edge?" in constant time; use it for dense graphs and matrix-style algorithms. **Table 10.5** has every cost and **Table 10.6** the decision.

Breadth-first search manages the frontier as a **queue**. It computes, for a source s, the distance $d[v] = \delta(s, v)$ to every vertex and a predecessor $\pi[v]$ from which a shortest path can be read off. It runs in $O(V + E)$, which is linear in the size of the input. Theorem 10.4 proves it correct, resting on Lemma 10.1 (one step cannot take you two steps further away), Lemma 10.2 (BFS never underestimates) and Lemma 10.3 (the queue holds at most two adjacent distances, in order). Its output is a **breadth-first tree**, a spanning tree of s's component in which every root-to-vertex path is a shortest path.

Depth-first search manages the frontier as a **stack** — usually the call stack. It takes no source and searches the whole graph, producing a **depth-first forest** and two timestamps per vertex, the discovery time $d[u]$ and the finishing time $f[u]$. It runs in $\Theta(V + E)$. Its structure theorems are the **parenthesis theorem** (any two vertices' intervals are nested or disjoint, never partially overlapping) and the **white-path theorem** (v becomes a descendant of u exactly when a path of white vertices runs from u to v at the moment u is discovered). Every edge is a **tree**, **back**,

forward or **cross** edge, decided by the color of its head when it is examined; in an undirected graph only tree and back edges occur (Theorem 10.9).

Six applications, all $O(V + E)$: *connected components* by repeated search; *cycle detection*, by any non-parent back edge when undirected and by an arc into a gray vertex when directed; *bipartiteness* by two-coloring during a BFS, which succeeds unless an edge joins two vertices at the same level; *shortest paths* in an unweighted graph, which is just BFS; the **DAG theorem**, that a directed graph is acyclic exactly when a DFS finds no back edge; and **topological sort**, which lists a DAG's vertices in decreasing order of finishing time and needs no sorting to do it.

The chapter's single most transferable idea is the one at the top of §10.6: **BFS and DFS differ in one line — the container — and everything else about them follows from that.**

10.12.1 Looking ahead

Chapter 11 puts weights on the edges and asks for the cheapest spanning tree. Both algorithms it gives — Prim's and Kruskal's — are greedy in the sense of Chapter 9, and both are graph searches in the sense of this one: Prim is breadth-first search with the queue replaced by a priority queue, and Kruskal is a sweep over the edge list of §10.5.2. Chapter 11 also returns to topological sort, gives Kahn's in-degree algorithm as the alternative to §10.9.6, and uses the topological order to solve shortest paths in a DAG in linear time.

Chapter 13 replaces "fewest edges" with "least total weight" and gives Dijkstra's algorithm, which is precisely the BFS of this chapter with `std::queue` changed to `std::priority_queue` — and then Bellman-Ford, for when some weights are negative and even that is not enough. Chapter 15 asks for the distance between *every* pair of vertices, and there the adjacency matrix of §10.5.4 finally becomes the right representation, because the algorithm is matrix arithmetic.

Everything in those four chapters assumes that you can traverse a graph in linear time and know why it is linear. That is what this chapter was for.

10.13 Exercises

10.1 For the graph G of **Figure 10.1**: (a) write out $\mathrm{Adj}[v]$ for all five vertices; (b) give $\deg(v)$ for each; (c) verify the handshake lemma numerically; (d) how many unordered pairs of distinct vertices are *not* adjacent, and which are they?

10.2 Prove that in any undirected graph the number of vertices of odd degree is even. Then decide, with a reason, whether a simple undirected graph can exist with degree sequence (3,3,3,3,3), and whether one can exist with (4,4,4,4,4).

10.3 In the graph of **Figure 10.1**, list **every** simple path from b to d, in order of increasing length. State $\delta(b, d)$ and say how you know it is the minimum.

10.4 A directed graph has $V = \{0,1,2,3\}$ and the edge list $[(0,1), (0,2), (1,2), (2,0), (2,3), (3,3)]$. (a) Draw it. (b) Give its adjacency lists. (c) Give its adjacency matrix. (d) Give $\deg^+(v)$ and $\deg^-(v)$ for each v and check both sums. (e) Count the machine words each of the three representations needs, as in **Table 10.4**.

10.5 For each situation, choose a representation from **Table 10.6** and justify it in one sentence. (a) A social network with 3×10^8 users, each with about 200 friends, on which you must run BFS to find degrees of separation. (b) A 40×40 distance table between cities, on which you will run an all-pairs shortest-path algorithm. (c) A file of 10^7 edges that you must sort by weight and then

process in order. (d) A graph of 5 000 vertices on which the inner loop of your algorithm asks "is (u, v) an edge?" about 10^9 times.

10.6 Run BFS on the graph of §10.7.4 with source w instead of s, using the same adjacency-list order. Produce a table in the format of **Table 10.7**, then give the final d and π arrays and the tree edges.

10.7 Show that for the graph of §10.7.4 with source s, the breadth-first **tree** is the same no matter how the adjacency lists are ordered — although the order of the queue is not. Then give a graph on four vertices where the tree genuinely does depend on the ordering, and exhibit two different trees.

10.8 Corollary 10.5 says that in an undirected graph every edge joins two levels that differ by at most one. Exactly which half of that statement survives in a **directed** graph, and which fails? Give a three-vertex directed graph in which an arc joins level 2 to level 0.

10.9 Run DFS on the directed graph of §10.8.4 but with the outer loop considering the vertices in the order z, y, x, w, v, u. Give the timestamps, the roots, the tree edges, the classification of all eight arcs, and the order of finishing. Compare the number of trees with **Table 10.9**.

10.10 Using only the timestamps of **Figure 10.25** — s 1/10, z 2/9, y 3/6, x 4/5, w 7/8, t 11/16, v 12/13, u 14/15 — decide, by Corollary 10.7 and without looking at the picture: (a) is x a descendant of s? (b) is w a descendant of y? (c) is v a descendant of u, or u of v? (d) is z a descendant of s? (e) is u a descendant of t?

10.11 Still using only those timestamps, classify each of these arcs of the same graph: $s \to w$, $x \to z$, $w \to x$, $u \to t$, $v \to s$. State the rule you used for each.

10.12 Give a directed graph on two vertices, and two orders for the outer loop of DFS, such that one order produces a forest with one tree and the other produces a forest with two. What does this say about the claim "DFS produces *the* depth-first forest of G"?

10.13 Decide which of these are bipartite, and for each that is not, exhibit an odd cycle: (a) the cycle C_7; (b) the complete bipartite graph $K_{3,3}$; (c) a tree on 9 vertices; (d) the graph of **Figure 10.1**; (e) the graph of **Figure 10.9**.

10.14 The topological sort of §10.9.6 produces *a* valid order. Design an algorithm that produces the **lexicographically smallest** valid order (treating the vertices as numbered $0 \ldots |V| - 1$), state its running time, and explain in one sentence why the DFS-based method of §10.9.6 cannot simply be adapted.

10.15 Modify BFS so that, in addition to $d[v]$, it computes $cnt[v]$, the **number of distinct shortest paths** from s to v. Give the modification in three lines, justify it, and apply it by hand to the graph of **Figure 10.1** with source a.

10.16 The **diameter** of a tree is the length of the longest simple path in it. Prove that the following computes it: run BFS from any vertex, let a be a furthest vertex found; run BFS from a, let b be a furthest vertex from a; the answer is $d[b]$. Then apply it to the tree with edges 0–1, 1–2, 2–3, 3–4, 2–5, 5–6, 1–7, 7–8, 8–9.

10.17 Write, in words and then as a five-line procedure, an $O(V + E)$ test for "is this undirected graph a tree?" that uses only a single BFS and a count. Explain why both of the conditions you test are necessary.

10.18 Profitina Laundry opens a new road between Semolowaru and Sukolilo. Recompute the BFS of **Listing 10.11** by hand, give every zone's new hop count, and say which zones enter the two-hop promise that were not in it before. Does Dukuh Setro become reachable?

10.14 Solutions to the Exercises

10.1 (a) Read them off the picture:

$$\text{Adj}[a] = \{b, c, d\},\ \text{Adj}[b] = \{a, e\},\ \text{Adj}[c] = \{a, d, e\},\ \text{Adj}[d] = \{a, c, e\},\ \text{Adj}[e] = \{b, c, d\}.$$

(b) $\deg(a) = 3$, $\deg(b) = 2$, $\deg(c) = 3$, $\deg(d) = 3$, $\deg(e) = 3$ — the five labels of **Figure 10.4**.

(c) $3 + 2 + 3 + 3 + 3 = 14$ and $2|E| = 2 \times 7 = 14$. They agree.

(d) There are $\binom{5}{2} = 10$ unordered pairs and 7 edges, so 3 pairs are non-adjacent: $\{a, e\}$, $\{b, c\}$, $\{b, d\}$. Check each against E and you will not find it.

10.2 *Proof.* Split V into V_{even} and V_{odd} by the parity of the degree. Then

$$\sum_{v \in V_{\text{even}}} \deg(v) + \sum_{v \in V_{\text{odd}}} \deg(v) = 2|E|,$$

which is even. The first sum is a sum of even numbers, hence even. Subtracting, the second sum is even. But a sum of odd numbers is even exactly when there is an even number of terms. So $|V_{\text{odd}}|$ is even. ■

(3,3,3,3,3): five vertices of odd degree, and five is odd — impossible by the result just proved. (Equivalently, the degrees sum to 15, which cannot equal $2|E|$.)

(4,4,4,4,4): the degrees sum to $20 = 2|E|$, so $|E| = 10$; and $10 = \binom{5}{2}$, so the graph would have to be K_5, in which every vertex indeed has degree 4. So yes, and the graph is unique: K_5.

10.3 There are six simple paths, and **Table 10.14** lists them all:

Table 10.14 All six simple paths from b to d in the graph of **Figure 10.1**, in order of increasing length. The shortest have length 2; there is none of length 1 because $(b, d) \notin E$.

Path	Length	Edges used
$b\,a\,d$	2	$(a, b), (a, d)$
$b\,e\,d$	2	$(b, e), (d, e)$
$b\,a\,c\,d$	3	$(a, b), (a, c), (c, d)$
$b\,e\,c\,d$	3	$(b, e), (c, e), (c, d)$
$b\,a\,c\,e\,d$	4	$(a, b), (a, c), (c, e), (d, e)$
$b\,e\,c\,a\,d$	4	$(b, e), (c, e), (a, c), (a, d)$

$\delta(b, d) = 2$. It cannot be 0, since $b \neq d$; it cannot be 1, since (b, d) is one of the three non-edges found in Exercise 10.1(d); and length 2 is achieved twice. By the reduction argument of §10.4.5 a shortest path is always simple, so this enumeration is exhaustive.

10.4 (a) Four vertices; arcs $0 \to 1$, $0 \to 2$, $1 \to 2$, $2 \to 0$, $2 \to 3$, and a self-loop at 3. (b) Adj[0] = [1,2], Adj[1] = [2], Adj[2] = [0,3], Adj[3] = [3]. (c) With rows and columns in the order 0,1,2,3:

$$M = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 1 \end{array}$$

It is not symmetric, because the graph is directed; and $M[3][3] = 1$ records the self-loop. (d) Out-degrees 2,1,2,1 summing to 6; in-degrees 1,1,2,2 summing to 6; and $|E| = 6$. Both sums equal $|E|$, as §10.4.4 requires. Vertex 0 has in-degree 1, so the graph has no source; vertex 3 has out-degree 1 (its own loop), so no sink either. (e) Edge list: $2 \times 6 = 12$ words. Adjacency list: 4 heads $+6$ entries (one per arc, since directed) $=10$ words. Adjacency matrix: $4^2 = 16$ words. The matrix is already the largest on four vertices.

10.5 (a) **Adjacency list.** $|E| \approx 3 \times 10^{10}$ against a matrix's 9×10^{16} entries, which cannot be stored; and BFS asks only Q3, which the list answers optimally. (b) **Adjacency matrix.** $40^2 = 1\,600$ entries is nothing, the graph is dense and weighted, and an all-pairs algorithm such as Floyd-Warshall (Chapter 15) *is* matrix arithmetic. (c) **Edge list.** The only operation is Q4, list all edges — and then a sort. This is Kruskal's pattern (Chapter 11). (d) **Adjacency matrix.** $5\,000^2 = 2.5 \times 10^7$ entries is about 25 MB at one byte each, which is affordable, and it turns 10^9 Q1 queries from $O(\deg u)$ each into $\Theta(1)$ each.

10.6 BFS from w, with $\mathrm{Adj}[w] = [s, t, x]$, runs as **Table 10.15** shows:

Table 10.15 Breadth-first search from w on the graph of §10.7.4. Compare with **Table 10.7**: the same eight steps, but a different tree and different distances, because the source has moved.

Step	Dequeue	Discovered, with d	Queue after
—	—	w: 0	(w: 0)
1	w	s: 1, t: 1, x: 1	(s: 1, t: 1, x: 1)
2	s	r: 2	(t: 1, x: 1, r: 2)
3	t	u: 2	(x: 1, r: 2, u: 2)
4	x	y: 2	(r: 2, u: 2, y: 2)
5	r	v: 3	(u: 2, y: 2, v: 3)
6	u	—	(y: 2, v: 3)
7	y	—	(v: 3)
8	v	—	( )

Final arrays: $d[w] = 0$; $d[s] = d[t] = d[x] = 1$; $d[r] = d[u] = d[y] = 2$; $d[v] = 3$. Predecessors: $\pi[s] = \pi[t] = \pi[x] = w$, $\pi[r] = s$, $\pi[u] = t$, $\pi[y] = x$, $\pi[v] = r$, $\pi[w] =$ NIL. Tree edges: $(w, s), (w, t), (w, x), (s, r), (t, u), (x, y), (r, v)$ — seven again, for eight vertices. Note that v is now at distance 3 rather than 2: distances are properties of the *pair*, not of the graph alone.

10.7 *Why the tree is forced here.* A vertex v with $d[v] = k$ can only be discovered from a neighbor at distance $k - 1$ (Theorem 10.4 plus Corollary 10.5), so the tree is unique exactly when every non-source vertex has exactly **one** neighbor one level closer. Check all seven, as **Table 10.16** does:

Table 10.16 Why the breadth-first tree of §10.7.4 is independent of the adjacency ordering: every vertex has exactly one neighbor one level closer to s, so it has no choice of parent.

v	$d[v]$	Neighbors, with their d	At level $d[v]-1$
r	1	s: 0, v: 2	s only
w	1	s: 0, t: 2, x: 2	s only
t	2	u: 3, w: 1, x: 2	w only
x	2	t: 2, w: 1, y: 3	w only
v	2	r: 1	r only
u	3	t: 2, y: 3	t only
y	3	u: 3, x: 2	x only

The *queue order* still changes — with $\mathrm{Adj}[s] = [r, w]$ the queue after step 1 is (r, w) rather than (w, r) — but no vertex ever has a choice of parent, so π is the same.

A graph where the tree is not forced. Take the 4-cycle C_4 on $\{0,1,2,3\}$ with edges 0–1, 1–2, 2–3, 3–0, and source 0. Then $d = (0,1,2,1)$ and vertex 2 has **two** neighbors at level 1, namely 1 and 3. With $\mathrm{Adj}[0] = [1,3]$ the search reaches 2 from 1 and the tree edges are $(0,1), (0,3), (1,2)$; with $\mathrm{Adj}[0] = [3,1]$ it reaches 2 from 3 and the tree edges are $(0,1), (0,3), (3,2)$. Two different trees, the same distances.

10.8 The surviving half is $\delta(s, v) \le \delta(s, u) + 1$ for every arc (u, v) — Lemma 10.1, whose proof never used symmetry: append the arc to a shortest path to u. What fails is the reverse inequality $\delta(s, u) \le \delta(s, v) + 1$, because you cannot traverse an arc backward. So in a directed graph an arc may drop from any level to any lower level, but may never rise by more than one.

Example: $V = \{s, p, q\}$ with arcs $s \to p$, $p \to q$, $q \to s$. BFS from s gives $d[s] = 0$, $d[p] = 1$, $d[q] = 2$. The arc $q \to s$ joins level 2 to level 0 — a drop of two. Its existence is consistent, because Lemma 10.1 applied to (q, s) only demands $\delta(s, s) \le \delta(s, q) + 1$, i.e. $0 \le 3$.

10.9 With the outer loop trying z, y, x, w, v, u, the timestamps come out as in **Table 10.17**:

Table 10.17 Timestamps when the DFS outer loop scans z, y, x, w, v, u. Compare **Table 10.9**, which used the order u, v, w, x, y, z: the same graph, the same algorithm, an entirely different forest.

v	u	v	w	x	y	z
d/f	11/12	5/6	9/10	4/7	3/8	1/2

Roots: z, y, w, u — **four** trees, where the original order gave two. Tree edges: only (y, x) and (x, v); the forest has $6 - 4 = 2$ edges, as the rule of §10.8.5 requires.

Classification of all eight arcs, in the order they are examined: $z \to z$ back; $y \to x$ tree; $x \to v$ tree; $v \to y$ back; $w \to y$ cross; $w \to z$ cross; $u \to v$ cross; $u \to x$ cross.

Order of finishing: z, v, x, y, w, u.

Two things are worth noticing. The arc $w \to z$ was a *tree* edge before and is a *cross* edge now, because z had already been searched as a root. And the back edges have changed identity — $x \to v$ was a back edge and is now a tree edge, while $v \to y$ is now the back edge — but a back edge still exists, as Theorem 10.10 guarantees it must, since the graph does contain a cycle.

10.10 By Corollary 10.7, v is a proper descendant of u exactly when $d[u] < d[v] < f[v] < f[u]$. (a) x a descendant of s: $1 < 4 < 5 < 10$. **Yes.** (b) w a descendant of y: needs $3 < 7 < 8 < 6$, and $8 < 6$ is false. The intervals [3,6] and [7,8] are disjoint, so by the parenthesis theorem neither is a descendant of the other. **No.** (c) [14,15] and [12,13] are disjoint, so **neither** is a descendant of the

other; they are siblings, both children of t, whose interval [11,16] contains both. (d) z a descendant of s: $1 < 2 < 9 < 10$. **Yes.** (e) u a descendant of t: $11 < 14 < 15 < 16$. **Yes.**

10.11 The rules, from **Table 10.11**: an arc (u, v) is a *back* edge when $d[v] < d[u] < f[u] < f[v]$ (v is an ancestor); a *forward* edge when $d[u] < d[v] < f[v] < f[u]$ and it is not the tree edge itself; a *cross* edge when $f[v] < d[u]$ (v's whole interval finished before u began).

- $s \to w$: $d[s] = 1 < d[w] = 7 < f[w] = 8 < f[s] = 10$. Nested, w a descendant of s, and w's parent is z not s — so **forward**.
- $x \to z$: $d[z] = 2 < d[x] = 4 < f[x] = 5 < f[z] = 9$. z is an ancestor of x — **back**.
- $w \to x$: $f[x] = 5 < d[w] = 7$. x's interval closed before w's opened — **cross**.
- $u \to t$: $d[t] = 11 < d[u] = 14 < f[u] = 15 < f[t] = 16$. t is an ancestor of u — **back**.
- $v \to s$: $f[s] = 10 < d[v] = 12$. Disjoint, s finished first — **cross**. (And indeed s and v are in different trees.)

All five agree with the labels in **Figure 10.25** and with the redrawing in **Figure 10.27**.

10.12 Take $V = \{0,1\}$ with the single arc $0 \to 1$. Outer order 0,1: DFS-Visit(0) discovers 1 as a tree edge, so the forest is **one** tree with edge (0,1). Outer order 1,0: DFS-Visit(1) finds Adj[1] empty and finishes; then DFS-Visit(0) examines $0 \to 1$, finds 1 black with $d[0] = 3 > d[1] = 1$, and calls it a cross edge. The forest is **two** single-vertex trees.

The moral is that there is no such thing as *the* depth-first forest of G; there is only the depth-first forest of G *under a given vertex order and a given adjacency order*. Every theorem in §10.8 is stated about "a depth-first search", not "the" one, for exactly this reason. What *is* invariant is the existence of a back edge whenever G has a cycle (Theorem 10.10) — that holds for every order, which is why cycle detection is a reliable answer while the forest is not.

10.13 (a) C_7 is an odd cycle, so **not bipartite**; the odd cycle is the whole graph, of length 7. (b) $K_{3,3}$ is **bipartite** by construction: the two sides are the two triples, and every edge crosses. (c) Any tree is **bipartite**. It is acyclic, so it has no cycle at all, in particular no odd one; concretely, color by the parity of the depth from any root. (d) The graph of **Figure 10.1** is **not bipartite**. The odd cycle is the triangle $a\ c\ d\ a$ of **Figure 10.7**. (**Listing 10.8** reports the failure at the edge c–d.) (e) The graph of **Figure 10.9** is **not bipartite**, because its first component is a triangle. Note that bipartiteness must hold in *every* component: one bad component spoils the whole graph.

10.14 Use **Kahn's algorithm** [3] with a priority queue. Compute every in-degree in $O(V + E)$. Put every vertex of in-degree 0 into a min-heap. Repeatedly extract the smallest available vertex, append it to the output, and decrement the in-degree of each of its out-neighbors, pushing any that reaches 0. If the output has fewer than $|V|$ vertices, the graph was not a DAG.

Cost. Each vertex is pushed and popped once, at $O(\log V)$ each, and each arc is relaxed once at $O(1)$ plus at most one push: $O((V + E)\log V)$.

Correctness of the greedy choice. At each step, the vertices of in-degree 0 among those not yet output are exactly the vertices that may legally come next. Choosing the smallest of them is safe: any valid order that placed a larger vertex y next could be rewritten by moving the smallest available x to the front of the remaining suffix — nothing before x in that suffix is required to precede it, since x's remaining in-degree is 0 — and the result is lexicographically smaller. So no lexicographically smaller order exists.

Why DFS cannot simply be adapted. The DFS method of §10.9.6 produces the order **backward**, by finishing times, so a greedy choice made at the front of the DFS is a choice about the *end* of the

output. Choosing the smallest vertex to finish first does not make the finished list smallest when reversed. There is no local rule at the DFS end that corresponds to "smallest first" at the output end.

10.15 Three lines. Keep an array cnt with $cnt[s] = 1$ and $cnt[v] = 0$ otherwise, and inside the loop over $v \in \text{Adj}[u]$ replace the body by the two lines of **Pseudocode 10.4**:

Pseudocode 10.4 The inner loop of BFS extended to count shortest paths: $cnt[v]$ accumulates the counts of all neighbors one level closer to the source.

```
for each v in Adj[u]
    if d[v] == UNSET
        d[v] = d[u] + 1;  cnt[v] = cnt[u];  ENQUEUE(Q, v)   // first shortest path found
    else if d[v] == d[u] + 1
        cnt[v] = cnt[v] + cnt[u]                            // one more family of shortest
paths
```

Justification. Every shortest path from s to v has length $d[v]$ and so arrives at v from some neighbor u with $d[u] = d[v] - 1$; conversely, prefixing any shortest path to such a u with the edge (u, v) gives a shortest path to v. So $cnt[v] = \sum\{cnt[u]: u \in \text{Adj}[v],\ d[u] = d[v] - 1\}$. The BFS visits every such u before v is dequeued — because vertices are dequeued in non-decreasing order of d — and each contributes its share exactly once, either when v is first discovered or in the `else if`. Each edge is still examined once, so the cost is still $O(V + E)$.

By hand, on **Figure 10.1** *with source a.* $d[a] = 0$, $cnt[a] = 1$. From a we discover b, c, d, each at distance 1 and each with $cnt = 1$. Then dequeue b: its neighbor e is white, so $d[e] = 2$ and $cnt[e] = cnt[b] = 1$. Dequeue c: e now has $d[e] = 2 = d[c] + 1$, so $cnt[e] += cnt[c]$, giving 2. Dequeue d: likewise $cnt[e] += cnt[d]$, giving 3. Final answer:

$$cnt[a] = 1, \quad cnt[b] = cnt[c] = cnt[d] = 1, \quad cnt[e] = 3.$$

And indeed the three shortest paths from a to e are $a\ b\ e$, $a\ c\ e$ and $a\ d\ e$.

10.16 *Proof.* Let the tree be T and let $P = p \cdots q$ be a longest path, of length D. Run BFS from any vertex r and let a be a furthest vertex from r. We claim a is an endpoint of *some* longest path. Two facts about trees do all the work: between any two vertices there is exactly one simple path, so δ is measured along it; and if a path leaves P at a vertex it cannot return to P, since that would close a cycle.

Let w be the vertex of P closest to r. Every path from r to a vertex of P reaches P first at w. There are two cases, according to whether the path from r to a touches P.

Case 1: the path from r to a touches P. It reaches P at w, runs along P for a while, and leaves P for good at some vertex z (possibly $z = w$, or $z = a$). By symmetry we may name the ends of P so that z lies between w and q. Then $\delta(r, a) = \delta(r, z) + \delta(z, a)$ and, because the path from r to q also passes through z, $\delta(r, q) = \delta(r, z) + \delta(z, q)$. Since a is a furthest vertex from r, $\delta(r, a) \geq \delta(r, q)$, and subtracting $\delta(r, z)$ gives $\delta(z, a) \geq \delta(z, q)$. Now walk from p along P to z and then out to a: this is a simple path, of length $\delta(p, z) + \delta(z, a) \geq \delta(p, z) + \delta(z, q) = D$. It cannot be longer than D, so it has length exactly D, and a is an endpoint of a longest path.

Case 2: the path from r to a does not touch P. Let y be the last vertex that the path from r to a shares with the path from r to w; then $y \neq w$, so $\delta(y, w) \geq 1$. As before, $\delta(r, a) \geq \delta(r, q)$ becomes $\delta(y, a) \geq \delta(y, w) + \delta(w, q)$. But then the simple path $p \cdots w \cdots y \cdots a$ has length $\delta(p, w) + \delta(w, y) + \delta(y, a) \geq \delta(p, w) + 2\,\delta(w, y) + \delta(w, q) = D + 2\,\delta(w, y) > D$, which is impossible because D is the longest. So Case 2 never happens.

The second BFS, from a, therefore finds a vertex b at distance exactly D, and D is the diameter. Two BFS runs, $O(V + E) = O(V)$ in a tree. ■

Applied. The tree has edges 0–1, 1–2, 2–3, 3–4, 2–5, 5–6, 1–7, 7–8, 8–9: ten vertices, nine edges, so it is a tree. BFS from 0 gives distances 0,1,2,3,4,3,4,2,3,4, so a furthest vertex is $a = 4$, at distance 4. BFS from 4 gives 4,3,2,1,0,3,4,4,5,6, so the furthest is $b = 9$ at distance 6. **The diameter is** 6, realized by the path 4 3 2 1 7 8 9. Note that the first BFS's answer, 4, was *not* the diameter — the second run is essential.

10.17 *In words.* An undirected graph is a tree exactly when it is **connected** and has exactly $|V| - 1$ **edges** (§10.4.11: any two of connected, acyclic, $|E| = |V| - 1$ imply the third). So: count the edges; run one BFS from vertex 0; check that it reached all $|V|$ vertices. **Pseudocode 10.5** is the procedure.

Pseudocode 10.5 Testing whether an undirected graph on $n \geq 1$ vertices is a tree: one edge count and one breadth-first search.

```
IS-TREE(adj, n)
if COUNT-EDGES(adj) != n - 1              // wrong number of edges
    return FALSE
d = BFS(adj, 0)                           // one search, O(V+E)
for v = 0 to n - 1
    if d[v] == UNSET                      // not connected
        return FALSE
return TRUE
```

Why both conditions are necessary. Neither implies the other. A triangle plus an isolated vertex has $|V| = 4$ and $|E| = 3 = |V| - 1$, but is **not connected** — so the edge count alone is not enough. A triangle alone has $|V| = 3$, is connected, but has $|E| = 3 > |V| - 1$ and contains a cycle — so connectivity alone is not enough. Together they force acyclicity: a connected graph with a cycle has at least $|V|$ edges, since deleting one edge of the cycle leaves it connected and any connected graph needs at least $|V| - 1$.

One special case to guard: for $n = 0$ the edge count test reads $0 \neq -1$ and returns false, and BFS from vertex 0 would be out of range. Handle the empty graph before anything else, or require $n \geq 1$.

10.18 The new road Semolowaru–Sukolilo makes Sukolilo, already at distance 1, a neighbor of Semolowaru; **Table 10.18** gives the before-and-after.

Table 10.18 The effect of one new road on Profitina Laundry's hop counts. One zone joins the two-hop promise and one leaves the four-hop tail; the zone with no road at all is unaffected, because adding an edge elsewhere cannot create a path to a different component.

Zone	Hops before	Hops after	Change
Keputih	0	0	—
Gebang Putih	1	1	—
Sukolilo	1	1	—
Klampis Ngasem	2	2	—
Medokan Semampir	2	2	—
Mulyorejo	2	2	—
Semolowaru	3	2	**improved**, now via Sukolilo
Manyar Sabrangan	3	3	—
Kalijudan	3	3	—
Nginden Jangkungan	4	3	improved, now via Semolowaru
Dukuh Setro	∞	∞	still unreachable

Semolowaru is the only zone that enters the two-hop promise, taking it from six zones to seven. Nginden Jangkungan improves from four hops to three but still misses the promise.

Dukuh Setro does not become reachable. It has no incident edge at all, so it is a connected component of its own (§10.4.8), and no edge added between two *other* zones can create a path into it. By fact 3 of §10.4.8, a search started in Keputih's component can never leave it. Only a road **touching Dukuh Setro** would change this — which is precisely the finding the laundry needs: the problem is not scheduling, it is that the zone is in a different component.

Where these ideas came from. The three-color scheme, the timestamps, the parenthesis and white-path theorems, the edge classification and the DFS-based topological sort are all presented here in the form given by Cormen, Leiserson, Rivest and Stein [1]; the two worked search examples are theirs. The labyrinth-with-a-string image for BFS, the paint-can extension for DFS, and the edge-list and adjacency-list pictures of §10.5 come from Goodrich and Tamassia [2]. Depth-first search as a general technique for linear-time graph algorithms is due to Tarjan [4], and the in-degree algorithm for topological sorting mentioned in Exercise 10.14 is Kahn's [3]. Dijkstra's paper [5] is the origin of the weighted shortest-path algorithm that Chapter 13 will present as BFS with a priority queue; Kleinberg and Tardos [6] give an unusually clear account of BFS and DFS as two disciplines for the same frontier; Sedgewick and Wayne [7] are the best source for implementation detail and for the many small applications of the two searches; Halim, Halim and Effendy [8] cover the competitive-programming variants including the 0-1 BFS mentioned in §10.9.4; and Knuth [9] gives the history of topological sorting and of the linked-list technique used in §10.9.6.

References

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

[2] M. T. Goodrich, R. Tamassia, and D. M. Mount, *Data Structures and Algorithms in C++*, 2nd ed. Hoboken, NJ, USA: Wiley, 2011.

[3] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962.

[4] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, Jun. 1972.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[6] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[7] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[8] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, 4th ed. Raleigh, NC, USA: Lulu Press, 2020.

[9] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

Chapter 11

Minimum Spanning Trees, DAGs, and Topological Sort

By the end of this chapter you will be able to look at a directed graph and decide, in linear time, whether it contains a cycle — and prove that your test is correct. You will be able to take a set of tasks with precedence constraints and produce an order in which they can legally be carried out, by either of the two standard methods, and say why a graph usually has many such orders rather than one. You will be able to state what a spanning tree is, prove that every spanning tree of a graph on $|V|$ vertices has exactly $|V| - 1$ edges, and explain why the *cheapest* spanning tree is a question worth asking. You will be able to state and prove the cut property — the single theorem on which every minimum-spanning-tree algorithm in existence rests — and then use it to prove two of them correct: Prim–Jarník, which grows one tree from a root, and Kruskal, which merges a forest from the cheapest edge upward. You will be able to trace each of them by hand on the chapter's example graph, printing the priority queue or the disjoint-set partition at every step. You will be able to implement a disjoint-set forest with union by rank and path compression, and say why its cost per operation is so close to constant that in practice you may treat it as constant. And you will be able to answer, when someone asks which of the two algorithms to use, with a reason rather than a preference.

11.1 Where We Left Off

The standard reference treatment of this material is Chapter 21 of Cormen, Leiserson, Rivest and Stein [1], and several figures of this chapter follow theirs closely. This chapter stands on two earlier chapters, and it is worth saying exactly what it takes from each.

From **Chapter 10** it takes the graph itself. A *graph* $G = (V, E)$ is a pair: a set V of **vertices**, and a set $E \subseteq V \times V$ of **edges**, each edge being a pair of vertices. If the pairs are ordered — if (u, v) and (v, u) mean different things — the graph is **directed**; if not, it is **undirected**. We write $|V|$ for the number of vertices and $|E|$ for the number of edges, and, as in Chapter 10, we drop the bars inside asymptotic notation: $O(V + E)$ always means $O(|V| + |E|)$. We represent a graph by its **adjacency lists**: $Adj[u]$ is the list of vertices that u points to (directed case) or is joined to (undirected case). Chapter 10 gave us two ways to walk a graph, breadth-first search and depth-first search, each in $O(V + E)$ time, and from depth-first search it gave us four pieces of bookkeeping that this chapter uses constantly.

> **DEFINITION (recalled from Chapter 10).** A depth-first search maintains a single **clock** that starts at 0 and ticks forward by one every time anything happens. For each vertex u it records a **discovery time** $d[u]$, the clock value at the moment u was first reached, and a **finishing time** $f[u]$, the clock value at the moment the search had finished examining every edge out of u and was about to back out of it. Always $d[u] < f[u]$. Each vertex is **white** before time $d[u]$ (not yet discovered), **gray** between $d[u]$ and $f[u]$ (discovered, but not yet finished with), and **black** after $f[u]$ (finished). The **predecessor** $\pi[u]$ is the vertex from which u was discovered, or NIL if u began a new tree of the depth-first forest.

Chapter 10 also classified the edges of a directed graph according to what the search found at the far end of each one:

Table 11.1 The four kinds of edge a depth-first search of a directed graph can traverse, classified by the color of the far endpoint at the moment the edge is examined. The whole of §11.2 turns on the second row.

Name	Color of v when (u,v) is explored	Meaning
tree edge	white	v is discovered through this edge; $\pi[v] = u$
back edge	gray	v is an ancestor of u in the depth-first forest
forward edge	black, and $d[u] < d[v]$	v is a descendant of u, reached another way first
cross edge	black, and $d[v] < d[u]$	everything else: between trees, or between subtrees

From **Chapter 9** this chapter takes the *discipline* of greedy design. A **greedy algorithm** builds a solution one piece at a time, and each time it has a choice it takes the piece that looks best at that moment, never reconsidering. That is a reckless-sounding way to solve an optimization problem, and Chapter 9 explained exactly what one has to prove before it is allowed.

> **DEFINITION (recalled from Chapter 9).** A problem has the **greedy-choice property** if there is always a globally optimal solution that contains the locally best choice. It has **optimal substructure** if, after that choice has been made, an optimal solution to the whole problem contains an optimal solution to the smaller problem that remains.

11.1.1 What is still missing

Chapter 10 walked graphs but never *measured* them. Every edge counted the same: one step. That is exactly right when the question is "can I get there?" or "how few edges away is it?", and it is exactly wrong when the question is "how much will it cost?". A road is long or short; a cable is cheap or dear; a courier run burns more petrol on one leg than another. The moment we attach a number to each edge we have a **weighted graph**, and a whole family of new questions opens up. This chapter answers one of them — *what is the cheapest set of edges that keeps everything connected?* — completely, from first principles, with two algorithms and a proof of each. Chapter 13 will answer a second, *what is the cheapest route from here to there?*, and Chapter 15 a third, *what is the cheapest route between every pair of places?*

Before any of that, though, there is an unfinished piece of Chapter 10 to clear away. Depth-first search left us holding a set of finishing times and no clear reason to care about them. This chapter opens by showing that those finishing times answer, all by themselves, a question that looks nothing like a search.

11.2 Directed Acyclic Graphs

11.2.1 Cycles, and what it means to have none

Start with the word *cycle*. In a directed graph, a **path** from v_0 to v_k is a sequence of vertices $v_0, v_1, \dots, v_k$ such that every consecutive pair is an edge: $(v_{i-1}, v_i) \in E$ for $i = 1,2,\dots,k$. The number k is the **length** of the path, counted in edges, not vertices. A path is a **cycle** if it has length at least one and it comes back to where it started, that is, $v_0 = v_k$.

> **DEFINITION.** A **directed acyclic graph**, universally abbreviated **DAG**, is a directed graph that contains no cycle. Equivalently: there is no vertex from which you can set off along the arrows and ever return to it.

Figure 11.1 shows two graphs, and both are DAGs. Follow the arrows from any vertex of the left graph and you run out of arrows; you never come back. The three-vertex graph on the right is the more instructive one, because it *looks* like a cycle: its three edges form a triangle. But read the directions. The top vertex points to both of the others, and the bottom-left vertex points to the bottom-right one. Start anywhere, follow the arrows, and you can never return — the bottom-right vertex has no way out at all. A triangle of edges is a directed cycle only if the arrows run round it in one direction. Reverse a single edge — make the bottom-right vertex point back up to the top — and the arrows now go top, bottom-left, bottom-right, top: a directed cycle of length three, and the graph is no longer a DAG. One such cycle is enough to disqualify a graph, no matter how large and well-behaved the rest of it is.

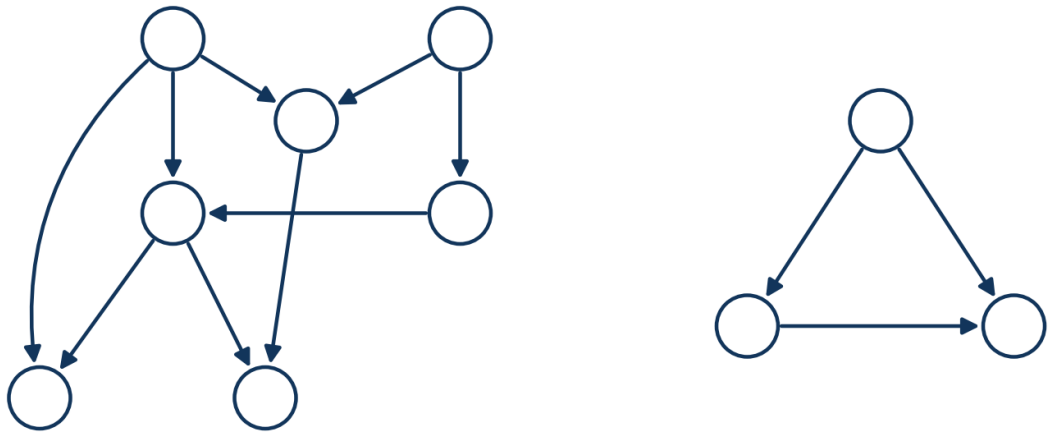


Figure 11.1 Two directed acyclic graphs. **Left:** follow the arrows from anywhere and you always run out of arrows. **Right:** three edges forming a triangle that is still acyclic, because the arrows do not run round it in one direction; reversing the edge from the top vertex to the bottom-right one would close a directed cycle.

> **IN PLAIN LANGUAGE.** A DAG is a set of things that must happen in some order, with no argument about it. "You must lay the foundation before the walls, and the walls before the roof" is a DAG. "You must ask the manager before you ask the cashier, and the cashier before the manager" is not a DAG — it is an impossible instruction, and the reason it is impossible is exactly that it contains a cycle.

11.2.2 Four everyday DAGs

The definition is dry. The reason DAGs matter is that this particular shape of graph turns up wherever one thing has to come before another. In one line: a DAG is often used to indicate precedences among events — event a must happen before event b. Four examples, none of them contrived.

Build dependencies. A program is built from source files. `main.o` needs `main.cpp` and every header it includes; `app` needs `main.o` and `graph.o`; `graph.o` needs `graph.cpp` and `graph.h`. Draw an arrow from each file to each file that depends on it and you have a DAG. `make` and every build system since does two things with that DAG: it checks that it really is acyclic (a cycle in a build graph is a bug — file A cannot need B while B needs A), and it finds an order in which to compile. When you see `make` reporting *circular dependency dropped*, you are watching a cycle-detection algorithm speak.

Course prerequisites. At ITS, *Design and Analysis of Algorithms* requires *Data Structures*, which requires *Programming Fundamentals*. Draw an arrow from each course to the courses it unlocks. If the graph is acyclic, a student can graduate; every legal study plan is an order on that DAG. If

somebody accidentally makes course X a prerequisite of Y and Y a prerequisite of X, the curriculum is unsatisfiable, and the cycle is the proof.

Task scheduling. A laundry job must be weighed before it is priced, sorted before it is washed, washed before it is dried, dried before it is pressed. Some pairs are unordered — weighing and sorting can happen in either order, or at the same time on two tables. A DAG records exactly the constraints that exist and nothing more, which is why it is the right model: it does not invent an order where none is required.

Parallel code execution. If instruction a writes a value that instruction b reads, a must run first; if two instructions touch nothing in common, they may run simultaneously on different cores. The dependency DAG of a block of code tells a compiler precisely which reorderings are legal, and how much parallelism is available.

In every one of those four, two questions come up immediately, and they are the two questions of this section and the next: **is it acyclic?** and, if so, **give me a legal order.**

11.2.3 The DAG theorem

Here is the theorem that makes the first question cheap to answer. Chapter 10 met it as Theorem 10.10; we restate it and prove it again here, in a form tuned to what follows.

> **THEOREM 11.1 (the DAG theorem).** A directed graph G is acyclic **if and only if** a depth-first search of G yields no back edges.

Read what that buys us. A back edge, by **Table 11.1**, is an edge (u, v) that a depth-first search examines at a moment when v is gray. Detecting gray is a single array lookup. So the theorem says: run one depth-first search, and the instant you examine an edge whose far end is gray, stop — the graph has a cycle. If you get to the end without that ever happening, the graph is acyclic. Since depth-first search costs $O(V + E)$ and we have added a constant amount of work per edge, **cycle detection costs** $O(V + E)$. That is as cheap as merely reading the graph, which is the best any algorithm could hope for.

The phrase *if and only if* means the theorem has two halves, and both need proving:

1. **If a back edge exists, then G has a cycle.** (A back edge is evidence *of* a cycle.)
2. **If G has a cycle, then a back edge exists.** (A cycle cannot hide from the search.)

The first half is easy. The second is the one that carries the weight, because it has to rule out the possibility that some cycle is arranged so cunningly that the search walks past it without noticing. Neither half may be skipped: half a proof would leave us with a test that is sound but possibly blind.

11.2.4 Proving the DAG theorem

Part 1: a back edge implies a cycle.

Suppose the search examines an edge (u, v) and finds v gray, so (u, v) is a back edge. By the definition of gray, v was discovered and has not finished. The gray vertices at any moment during a depth-first search are exactly the vertices on the current recursion stack — the chain of calls from the root of the current depth-first tree down to the vertex being worked on now. (Chapter 10 called this "the gray vertices form a linear chain".) Since u is the vertex being worked on, and v is gray, v sits somewhere on that chain at or above u. That is to say: v **is an ancestor of** u **in the depth-first forest**, or $v = u$ itself.

An ancestor relationship in the depth-first forest is built out of tree edges, and every tree edge is a real edge of G. So there is a path in G from v down to u:

$$v \rightsquigarrow u.$$

Now add the edge (u, v) to the end of that path. We arrive back at v. That is a closed walk of length at least one, which is a cycle. (If $v = u$, the edge (u, u) is itself a cycle of length one — a self-loop.) Therefore G is not acyclic. ■

Part 2: a cycle implies a back edge.

This is the direction that needs care, and the proof turns on one idea, which Chapter 10 called the **white-path property**: at the moment a vertex v is discovered, every vertex reachable from v along a path of entirely white vertices will become a descendant of v in the depth-first tree before v finishes.

Suppose G contains a cycle c. Let v be the **first vertex of c that the depth-first search discovers**. There must be a first one, since the search discovers every vertex eventually and the clock orders the discoveries. Let u be the predecessor of v *on the cycle*, so $(u, v) \in E$ and u lies on c. We will show that (u, v) is a back edge.

Consider the instant v is discovered, that is, clock time $d[v]$. Because v is the first vertex of c to be discovered, **every other vertex of c is still white at that instant** — including u. Now walk around the cycle from v in the direction of the arrows:

$$v \to v_1 \to v_2 \to \cdots \to u.$$

That is a path from v to u, and at time $d[v]$ every vertex on it after v is white. It is a white path. By the white-path property, the search cannot back out of v until it has visited every vertex reachable from v on white paths; so u becomes a descendant of v in the depth-first tree, and u is discovered and finished strictly inside the interval $(d[v], f[v])$.

Now: at some moment inside that interval the search stands at u and examines u's edges, one of which is (u, v). What color is v then? The clock is at some time t with $d[v] < t < f[v]$, and by the color rule of §11.1 a vertex is gray exactly between its discovery and its finishing time. So v is gray. An edge examined from a gray vertex to a gray vertex is, by **Table 11.1**, a back edge. Hence (u, v) is a back edge, and the search does find one. ■

> **IN PLAIN LANGUAGE.** Part 2 says a cycle cannot hide. Pick the vertex of the cycle you bump into first. At that instant the whole rest of the cycle is unexplored, so it is a clear white road leading back to your own doorstep. Depth-first search always walks a clear white road to its end before turning back — so it will walk all the way around the cycle and find itself knocking on the door of a room it is still standing in. That knock is the back edge.

11.2.5 Why the "first vertex" choice matters

Students often try Part 2 by picking an arbitrary vertex of the cycle, and the proof breaks. Here is why the word *first* is load-bearing.

Take the cycle $x \to y \to z \to x$ and suppose the search happens to discover y first, then finishes y and z entirely, and only later — from some unrelated vertex — discovers x. If we had picked x as our starting vertex and looked at the path $x \to y$, we would have found y already **black**, not white. The edge (x, y) is then a forward or cross edge, not a back edge, and the argument collapses.

Picking the *first-discovered* vertex v of the cycle removes that possibility by construction: no other vertex of the cycle can be anything but white at time $d[v]$, because "first discovered" means precisely that none of them has been discovered yet. In the example, the first-discovered vertex is y; the predecessor of y on the cycle is x; and it is (x, y) — examined later, from the descendant x, while y is still gray — that is the back edge. The proof finds the back edge wherever it is, but only if you let it choose which edge to look at.

A TRAP. Theorem 11.1 is about *directed* graphs, and the four-way edge classification it uses is a directed-graph classification. In an **undirected** graph the theorem as stated is false in a boring way: every edge $\{u, v\}$ appears in both $Adj[u]$ and $Adj[v]$, so the search will examine (v, u) immediately after coming to v from u and find u gray. That is not a cycle; it is the same edge seen twice. Undirected cycle detection needs the extra rule "ignore the edge you arrived on", and once you add it, a back edge again means a cycle. Exercise 11.3 asks you to get this right.

11.3 Topological Sort

11.3.1 What we want

We now have the second question: given a DAG of precedences, produce an order in which the tasks can legally be done. The definition:

DEFINITION. A **topological sort** of a directed acyclic graph $G = (V, E)$ is a linear ordering of all its vertices such that for every edge $(u, v) \in E$, the vertex u appears **before** the vertex v in the ordering.

Unpack every word of that. *Linear ordering* means a list containing each vertex exactly once — not a tree, not a set of groups, a single sequence. *All* its vertices means isolated vertices are in the list too, even though nothing constrains them. And the condition is a condition on every edge: if you lay the vertices out left to right in the order produced, **every arrow must point rightwards.** Not one arrow may point back. That is the whole specification, and it is worth carrying that picture in your head, because it is also how you check an answer.

IN PLAIN LANGUAGE. A topological sort is a to-do list you can actually work through from the top without ever being blocked. Every time you reach an item, everything it depended on is already crossed off.

The first thing to establish is that such an order exists at all. It does, and the reason is Theorem 11.1 turned around: a graph with a cycle has no topological order (if u comes before v and v comes before u, which does a cycle demand?), and — as the two algorithms below will constructively prove — a graph without one always has at least one.

11.3.2 Why a topological order is (almost) never unique

Beginners expect one answer. There is usually a great many. The reason is simple: the definition constrains only the pairs joined by an edge. Two vertices with no path between them in either direction may appear in the output in either order, and both outputs are correct.

Take the tiniest example: $V = \{a, b\}$, $E = \varnothing$. Both a, b and b, a are topological orders. Take $V = \{a, b, c\}$ with the single edge (a, b). Then c may go anywhere: c, a, b and a, c, b and a, b, c are all valid, and those are only three of them.

How many are there in general? For a DAG that is a single chain $v_1 \to v_2 \to \cdots \to v_n$, exactly one: every pair is constrained. For a DAG with no edges at all, exactly $n!$: nothing is constrained. Real

graphs sit between, and the count is the number of **linear extensions** of the partial order the DAG defines — a quantity that is genuinely hard to compute (it is #P-complete), which is itself a hint that there are usually a lot of them.

> **A TRAP.** Because the answer is not unique, you cannot check a topological sort by comparing it against a single stored answer, and you cannot test an implementation by comparing its output to a stored string. You must check the *property*: lay the output out, number the positions, and verify that $\text{pos}[u] < \text{pos}[v]$ for every edge (u, v). Both programs in this section end by doing exactly that, and so should yours.

We will see the non-uniqueness for free: the two algorithms below, run on the same graph, produce two completely different orders, and both are right.

11.3.3 The running example: getting dressed

Figure 11.2 is the running example, and it is the best small DAG in the literature because everybody has already solved it by hand this morning. Nine items of clothing. An arrow from x to y means *you must finish with x before you can do y*.

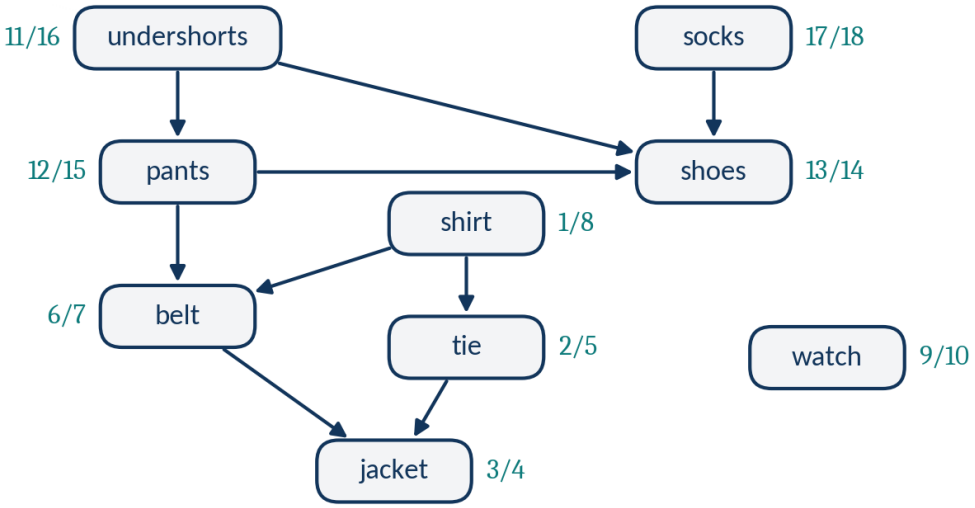


Figure 11.2 The precedence DAG for getting dressed, with the depth-first search timestamps $d[v]/f[v]$ written beside each vertex. Nine vertices, nine edges. `watch` has no edges at all: nothing constrains when you put it on.

Read the nine edges off the figure:

Table 11.2 The nine precedence constraints of **Figure 11.2**, read off the arrows. Note what is *not* here: nothing relates `socks` to `shirt`, and nothing at all relates `watch` to anything. Those freedoms are why the answer is not unique.

Edge	In words
undershorts → pants	undershorts go on before trousers
undershorts → shoes	and before shoes
pants → shoes	trousers before shoes
pants → belt	trousers before the belt
shirt → belt	shirt before the belt (it is tucked in)
shirt → tie	shirt before the tie
belt → jacket	belt before the jacket
tie → jacket	tie before the jacket
socks → shoes	socks before shoes

Is it a DAG? It must be, or getting dressed would be impossible. We will confirm it mechanically in a moment: the program reports `back edge found: no`.

11.3.4 Algorithm 1: depth-first search, and the finishing times

The algorithm, **Pseudocode 11.1**, is three lines long, and the whole of its cleverness is in line 2.

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
```

Note what it does *not* do: it does not sort anything, in the sense of comparing pairs. It runs one depth-first search and pushes each vertex onto the front of a list at the moment the search finishes with it. Since pushing onto the front of a linked list costs $O(1)$, the list ends up holding the vertices in **decreasing order of finishing time**, and — this is the claim — that order is a topological order.

Why would decreasing finishing time be the right order? Here is the intuition before the proof. A vertex finishes only when everything reachable from it has finished. So the things that depend on u all finish *before* u does. If we then list vertices from the latest finisher to the earliest, u comes out ahead of everything that depends on it. Which is what the definition asks for.

11.3.5 The example traced, one timestamp at a time

Run the search with the vertices considered, in the outer loop, in the order `shirt, tie, jacket, belt, watch, undershorts, pants, shoes, socks`, and with each adjacency list in the order given in **Table 11.2**. Here is every tick of the clock.

Table 11.3 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`	$d[\text{shirt}] = 1$; `shirt` gray
2	from `shirt`, `tie` is white: discover it	$d[\text{tie}] = 2$
3	from `tie`, `jacket` is white: discover it	$d[\text{jacket}] = 3$
4	`jacket` has no out-edges: finish it	$f[\text{jacket}] = 4$; list = `jacket`
5	back at `tie`, no more edges: finish it	$f[\text{tie}] = 5$; list = `tie, jacket`
6	back at `shirt`, `belt` is white: discover it	$d[\text{belt}] = 6$
7	from `belt`, `jacket` is **black** — nothing to do; finish `belt`	$f[\text{belt}] = 7$; list = `belt, tie, jacket`
8	back at `shirt`, no more edges: finish it	$f[\text{shirt}] = 8$
9	outer loop: `watch` is white, discover it	$d[\text{watch}] = 9$
10	`watch` has no out-edges: finish it	$f[\text{watch}] = 10$
11	outer loop: `undershorts` is white, discover it	$d[\text{undershorts}] = 11$
12	from `undershorts`, `pants` is white: discover it	$d[\text{pants}] = 12$
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`	$f[\text{pants}] = 15$
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`	$f[\text{socks}] = 18$

Now read the finishing times off, largest first:

$$\underbrace{18}_{\text{socks}},\ \underbrace{16}_{\text{undershorts}},\ \underbrace{15}_{\text{pants}},\ \underbrace{14}_{\text{shoes}},\ \underbrace{10}_{\text{watch}},\ \underbrace{8}_{\text{shirt}},\ \underbrace{7}_{\text{belt}},\ \underbrace{5}_{\text{tie}},\ \underbrace{4}_{\text{jacket}}.$$

That is the answer, and **Figure 11.3** pictures it: the same nine vertices laid out in a line, with every one of the nine arrows now curving *forward*, left to right, and not one curving back.

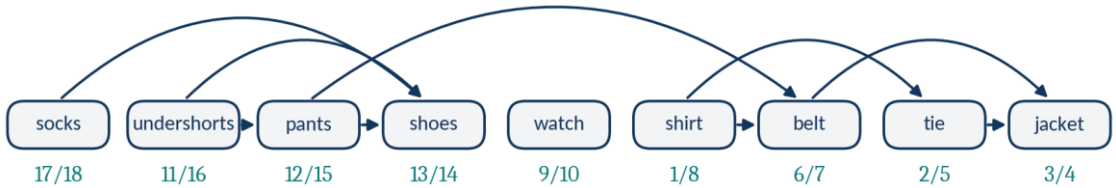


Figure 11.3 The result: the nine vertices of **Figure 11.2** laid out in decreasing order of finishing time. Every arrow now points to the right, which is exactly the definition of a topological sort. The d/f pair is printed under each vertex so the ordering can be checked against **Table 11.3**.

Check it by hand against **Table 11.2**. `undershorts` (position 2) before `pants` (3) and before `shoes` (4): yes. `pants` before `shoes` and before `belt` (7): yes. `shirt` (6) before `belt` (7) and `tie` (8): yes. `belt` (7) before `jacket` (9): yes. `tie` (8) before `jacket` (9): yes. `socks` (1) before `shoes` (4): yes. All nine.

Notice also how strange the answer looks as advice. It tells you to put your socks on first and your shoes fourth — before your shirt. That is *legal*: nothing in **Table 11.2** says a shirt must precede

shoes. It is merely not what you would do. A topological sort respects the constraints you gave it and nothing else, which is a feature; if you want shoes late, add an edge that says so.

11.3.6 Why it is correct

The proof is a three-case argument. Here it is in full.

> **THEOREM 11.2.** Let G be a DAG. Then for every edge $(u, v) \in E$ we have $f[v] < f[u]$. Consequently, listing the vertices in decreasing order of f is a topological sort of G.

Proof. Consider any edge (u, v), and consider the instant at which the depth-first search examines it. At that instant the search is standing at u, so u is gray. There are exactly three possibilities for the color of v, and we take them one at a time.

Case 1: v is gray. Then (u, v) is a back edge, and by Part 1 of Theorem 11.1 the graph contains a cycle. But G is a DAG. Contradiction — so this case cannot occur. (This is the only place the acyclicity hypothesis is used, and it is indispensable: on a cyclic graph the theorem is false, as it must be.)

Case 2: v is white. Then v is discovered through this very edge, so v becomes a child of u in the depth-first tree. The search must finish v before it can back out of u, because the recursive call on v is nested inside the call on u. Hence $f[v] < f[u]$.

Case 3: v is black. Then v has already finished, at some clock time strictly earlier than now. And u has not finished — it is gray — so $f[u]$ is some time strictly later than now. Hence $f[v] < \text{now} < f[u]$, giving $f[v] < f[u]$.

In every possible case, $f[v] < f[u]$. So in the list sorted by decreasing f, the vertex u appears before v. Since (u, v) was an arbitrary edge, this holds for every edge, which is precisely the definition of a topological sort. ■

> **IN PLAIN LANGUAGE.** Whatever u points at either finishes inside u's own visit (case 2) or had already finished before u started caring (case 3). Either way it finishes first. So "last to finish" is "furthest upstream", and sorting by last-to-finish puts the upstream things first.

11.3.7 The running time is $\Theta(V + E)$

The accounting takes two lines, and they are worth spelling out.

- The depth-first search costs $\Theta(V + E)$. The two loops over vertices cost $\Theta(V)$; `DFS-Visit` is called exactly once per vertex (it is invoked only on white vertices and paints its argument gray immediately, so no vertex is visited twice); and across all those calls the inner loop runs over $Adj[u]$ once for each u, and $\sum_u |Adj[u]| = |E|$ for a directed graph. Total: $\Theta(V + E)$.
- Inserting a vertex at the front of a linked list costs $O(1)$, and we do it once per vertex, so $O(V)$ in total.

Adding: $\Theta(V + E) + O(V) = \Theta(V + E)$. Since any algorithm must at least look at the input, and the input has size $\Theta(V + E)$ in the adjacency-list representation, this is optimal to within a constant factor. There is nothing left to improve.

> **A TRAP.** The insertion must be at the **front**, and it must happen at **finishing** time, not at discovery time. Appending to the back gives *increasing* finishing time, which is the topological order reversed — every arrow points the wrong way. And recording at discovery time gives the order the search happened to walk, which is not a topological order at all: in **Table 11.3** the discovery order begins `shirt, tie, jacket`, and `jacket` has two predecessors that have not appeared yet.

11.3.8 The complete program

Listing 11.1 is the whole thing, ready to compile. It carries the colors, the clock, d, f and π explicitly, so the printed timestamps can be compared line by line with **Table 11.3**; it watches for a gray-to-gray edge, so it also answers the cycle question of §11.2; and it finishes by verifying the topological property rather than trusting it.

Listing 11.1 Topological sort by depth-first finishing times, on the dressing DAG. Compile with `g++ -std=c++17 -O2 -o toposort toposort.cpp`.

```
// Topological sort by DFS finishing times.
// Reproduces the "getting dressed" example of Section 11.3.3.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <list>
using namespace std;

enum Color { WHITE, GRAY, BLACK };

struct Graph {
    int n;                          // number of vertices
    vector<string>       name;      // name[u] - printable name of vertex u
    vector<vector<int>>  adj;       // adj[u]  - the vertices u points to
    explicit Graph(const vector<string>& names)
        : n((int)names.size()), name(names), adj(names.size()) {}
    void addEdge(int u, int v) { adj[u].push_back(v); }
};

struct DfsState {
    vector<Color> color;
    vector<int>    d, f, pi;
    int            clock_ = 0;      // the monotonically increasing global clock
    list<int>      order;           // the linked list of TOPOLOGICAL-SORT, line 2
    bool           hasBackEdge = false;
};

void dfsVisit(const Graph& G, DfsState& S, int u) {
    S.color[u] = GRAY;
    S.d[u] = ++S.clock_;                      // u is discovered
    for (int v : G.adj[u]) {
        if (S.color[v] == GRAY) S.hasBackEdge = true;   // gray -> gray is a back edge
        if (S.color[v] == WHITE) { S.pi[v] = u; dfsVisit(G, S, v); }
    }
    S.color[u] = BLACK;
    S.f[u] = ++S.clock_;                      // u is finished
    S.order.push_front(u);                    // insert onto the FRONT of the list
}
```

Listing 11.1 (Continued) *Topological sort by depth-first finishing times, on the dressing DAG. Compile with g++ -std=c++17 -O2 -o toposort toposort.cpp.*

```
DfsState topologicalSort(const Graph& G) {
    DfsState S;
    S.color.assign(G.n, WHITE);
    S.d.assign(G.n, 0); S.f.assign(G.n, 0); S.pi.assign(G.n, -1);
    for (int u = 0; u < G.n; ++u)
        if (S.color[u] == WHITE) dfsVisit(G, S, u);
    return S;
}

int main() {
    // Index order fixes the order of the outer loop, and so the timestamps.
    vector<string> names = { "shirt", "tie", "jacket", "belt", "watch",
                             "undershorts", "pants", "shoes", "socks" };
    enum { SHIRT, TIE, JACKET, BELT, WATCH, UNDERSHORTS, PANTS, SHOES, SOCKS };
    Graph G(names);
    G.addEdge(SHIRT, TIE);        G.addEdge(SHIRT, BELT);
    G.addEdge(TIE, JACKET);
    G.addEdge(BELT, JACKET);
    G.addEdge(UNDERSHORTS, PANTS); G.addEdge(UNDERSHORTS, SHOES);
    G.addEdge(PANTS, SHOES);      G.addEdge(PANTS, BELT);
    G.addEdge(SOCKS, SHOES);
    // "watch" has no edges at all.

    DfsState S = topologicalSort(G);

    cout << "timestamps (d/f) in the order the DFS assigned them\n";
    vector<int> byDiscovery(G.n);
    for (int u = 0; u < G.n; ++u) byDiscovery[u] = u;
    for (int i = 0; i < G.n; ++i)                       // simple selection by d[]
        for (int j = i + 1; j < G.n; ++j)
            if (S.d[byDiscovery[j]] < S.d[byDiscovery[i]]) swap(byDiscovery[i],
byDiscovery[j]);
    for (int u : byDiscovery)
        cout << "  " << left << setw(12) << G.name[u] << right
             << setw(3) << S.d[u] << "/" << S.f[u] << "\n";

    cout << "back edge found: " << (S.hasBackEdge ? "yes" : "no") << "\n";
    cout << "topological order: ";
    for (int u : S.order) cout << G.name[u] << " ";
    cout << "\n";
```

Listing 11.1 (Continued) *Topological sort by depth-first finishing times, on the dressing DAG. Compile with g++ -std=c++17 -O2 -o toposort toposort.cpp.*

```
    cout << "check: every edge points forward\n";
    vector<int> pos(G.n); int k = 0;
    for (int u : S.order) pos[u] = k++;
    bool ok = true;
    for (int u = 0; u < G.n; ++u)
        for (int v : G.adj[u])
            if (pos[u] >= pos[v]) { ok = false;
                cout << "  VIOLATED: " << G.name[u] << " -> " << G.name[v] << "\n"; }
    cout << (ok ? "  all 9 edges satisfied" : "  order is not topological") << "\n";
    return 0;
}
```

Its output:

```
timestamps (d/f) in the order the DFS assigned them
  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
back edge found: no
topological order: socks undershorts pants shoes watch shirt belt tie jacket
check: every edge points forward
  all 9 edges satisfied
```

Every timestamp matches the number written beside that vertex in **Figure 11.2**, and the order matches **Figure 11.3** exactly.

11.3.9 Algorithm 2: Kahn's in-degree algorithm

The second algorithm is older than the first — A. B. Kahn published it in 1962 [2] — and it is the one most people invent for themselves if you ask them to write a to-do list from a dependency graph. It works from the front rather than the back.

> **DEFINITION.** The **in-degree** of a vertex v, written $\text{indeg}(v)$, is the number of edges that point *at* it: $\text{indeg}(v) = |\{u : (u, v) \in E\}|$. Its **out-degree** is the number pointing away.

A vertex of in-degree 0 has nothing waiting on it — no prerequisite, nothing upstream. So it is safe to do **now**. That single observation is the whole algorithm, **Pseudocode 11.2**:

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
```

Two details are worth pausing on. First, line 5 says *some* vertex: any ready vertex will do, and the choice is where all the non-uniqueness lives. Use a stack and you get one order; a queue, another; a min-heap on the vertex index, another again; and every one of them is correct. Second, lines 11–12 are a cycle test that comes free. If the graph has a cycle, no vertex on that cycle can ever reach in-degree 0 — each is waiting on its predecessor around the loop, forever — so those vertices, and anything downstream of them, never enter R and never reach L. The shortfall is the diagnosis.

11.3.10 Kahn traced on the same DAG

Start by counting arrowheads in **Figure 11.2**:

$$\text{indeg:shirt} = 0,\ \text{tie} = 1,\ \text{jacket} = 2,\ \text{belt} = 2,\ \text{watch} = 0,\ \text{undershorts} = 0,\ \text{pants} = 1,\ \text{shoes} = 3,\ \text{socks} = 0.$$

So $R = \{\text{shirt,watch,undershorts,socks}\}$ — four things you may do first. (Compare the DFS algorithm, which offered no such choice; here the freedom is visible from line one.) Taking the ready vertex of smallest index each time, the run goes as follows.

Table 11.4 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)

The order produced is

$$\text{shirt, tie, watch, undershorts, pants, belt, jacket, socks, shoes,}$$

which is nothing like the depth-first answer in **Figure 11.3**. Check it against **Table 11.2** anyway: `undershorts` (4) before `pants` (5) and `shoes` (9) — yes; `pants` (5) before `shoes` (9) and `belt` (6) — yes; `shirt` (1) before `belt` (6) and `tie` (2) — yes; `belt` (6) and `tie` (2) before `jacket` (7) — yes; `socks` (8) before `shoes` (9) — yes. Nine for nine. **Both answers are correct, and they are not the same answer.** That is §11.3.2 made concrete.

11.3.11 Why Kahn is correct

There are two things to prove: that what comes out is a topological order, and that everything comes out (when the graph is acyclic).

The output is a topological order. Maintain this invariant: *a vertex is appended to L only after every one of its predecessors has been appended.* It holds at the start, when L is empty. It is preserved by line 6: the vertex u removed from R has $\text{indeg}[u] = 0$ at that moment, and the counter for u started at its true in-degree and is decremented exactly once per predecessor appended (line 8). So $\text{indeg}[u] = 0$ means *all* of u's predecessors are already in L. Hence for every edge (p, u), p precedes u in L — the definition of a topological sort. ■

Everything comes out, if G is acyclic. Suppose not: the loop halts with R empty and some vertices left over. Let X be the set of left-over vertices; X is not empty. Every $v \in X$ has $\text{indeg}[v] > 0$ at the end, so v has at least one predecessor not in L — that is, a predecessor in X. So from any vertex of X we can step backward to another vertex of X, and from that one backward again, and so on forever. But X is finite, so some vertex must repeat, and a repeated vertex on a backward walk is a directed cycle. That contradicts acyclicity. Hence $X = \varnothing$ and $|L| = |V|$. ■

The same argument read forward is the cycle *detector*: if $|L| < |V|$, the leftover set X is non-empty and the argument above manufactures a cycle inside it. So the test of line 11 is not a heuristic; it is exact.

11.3.12 Kahn's running time

- Line 1 computes all in-degrees by sweeping every adjacency list once: $\Theta(V + E)$.
- Line 2 scans the vertices once: $\Theta(V)$.
- The `while` loop body runs once per vertex, since each vertex enters R at most once (it enters only when its counter *reaches* 0, and counters never rise). Each iteration does $O(1)$ work plus one pass over $Adj[u]$. Summed over all u: $\Theta(V + E)$.
- Lines 11–12 are $O(1)$.

Total $\Theta(V + E)$ — the same as the depth-first method, and again optimal. If R is a stack or a plain queue, every operation on it is $O(1)$ and the bound is exactly as stated; if you use a heap to make the output deterministic, as the program below does, each operation costs $O(\lg V)$ and the total becomes $O(V \lg V + E)$, which is a price paid purely for tidiness.

11.3.13 The complete program, with cycle detection

Listing 11.2 is the whole algorithm, with the ready set held in a min-heap so that the printed order is reproducible, and with a second run on a graph that has been deliberately broken so the cycle test can be seen working.

Listing 11.2 Kahn's in-degree topological sort, with the free cycle test. Compile with `g++ -std=c++17 -O2 -o kahn kahn.cpp`.

```
// Topological sort by in-degrees (Kahn's algorithm), on the same DAG,
// plus a cycle test on a graph that is deliberately not acyclic.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
using namespace std;

struct Graph {
    int n;
    vector<string>      name;
    vector<vector<int>> adj;
    explicit Graph(const vector<string>& names)
        : n((int)names.size()), name(names), adj(names.size()) {}
    void addEdge(int u, int v) { adj[u].push_back(v); }
};

// Returns the vertices in the order they were emitted. If that order is
// shorter than G.n the graph has a cycle: the vertices left out are exactly
// those that never reached in-degree 0, which is the cycle and whatever it feeds.
// `trace` prints one line per removal, showing what each removal frees.
vector<int> kahn(const Graph& G, bool trace) {
    vector<int> indeg(G.n, 0);
    for (int u = 0; u < G.n; ++u)
        for (int v : G.adj[u]) ++indeg[v];

    // A min-heap on the index makes the output deterministic; any container works.
    priority_queue<int, vector<int>, greater<int>> ready;
    for (int u = 0; u < G.n; ++u) if (indeg[u] == 0) ready.push(u);

    if (trace) {
        cout << "  initial in-degrees:";
        for (int u = 0; u < G.n; ++u) cout << " " << G.name[u] << "=" << indeg[u];
        cout << "\n";
    }
```

Listing 11.2 (Continued) *Kahn's in-degree topological sort, with the free cycle test. Compile with g++ -std=c++17 -O2 -o kahn kahn.cpp.*

```
    vector<int> order;
    int step = 0;
    while (!ready.empty()) {
        int u = ready.top(); ready.pop();
        order.push_back(u);
        vector<int> freed;
        for (int v : G.adj[u]) if (--indeg[v] == 0) { ready.push(v); freed.push_back(v); }
        if (trace) {
            cout << "  step " << setw(2) << ++step << ": emit " << left << setw(12)
                 << G.name[u] << right << " freed:";
            if (freed.empty()) cout << " -";
            for (int v : freed) cout << " " << G.name[v];
            cout << "\n";
        }
    }
    return order;
}

int main() {
    vector<string> names = { "shirt", "tie", "jacket", "belt", "watch",
                             "undershorts", "pants", "shoes", "socks" };
    enum { SHIRT, TIE, JACKET, BELT, WATCH, UNDERSHORTS, PANTS, SHOES, SOCKS };
    Graph G(names);
    G.addEdge(SHIRT, TIE);         G.addEdge(SHIRT, BELT);
    G.addEdge(TIE, JACKET);        G.addEdge(BELT, JACKET);
    G.addEdge(UNDERSHORTS, PANTS); G.addEdge(UNDERSHORTS, SHOES);
    G.addEdge(PANTS, SHOES);       G.addEdge(PANTS, BELT);
    G.addEdge(SOCKS, SHOES);

    cout << "Kahn on the dressing DAG\n";
    vector<int> ord = kahn(G, true);
    cout << "  order: ";
    for (int u : ord) cout << G.name[u] << " ";
    cout << "\n";

    vector<int> pos(G.n); for (int i = 0; i < (int)ord.size(); ++i) pos[ord[i]] = i;
    bool ok = true;
    for (int u = 0; u < G.n; ++u)
        for (int v : G.adj[u]) if (pos[u] >= pos[v]) ok = false;
    cout << "  valid topological order: " << (ok ? "yes" : "no") << "\n\n";
```

Listing 11.2 (Continued) *Kahn's in-degree topological sort, with the free cycle test. Compile with g++ -std=c++17 -O2 -o kahn kahn.cpp.*

```
    // Now break it. Adding jacket -> undershorts closes the directed cycle
    // undershorts -> pants -> belt -> jacket -> undershorts.
    cout << "Kahn on the same graph plus the edge jacket -> undershorts\n";
    Graph H = G;
    H.addEdge(JACKET, UNDERSHORTS);
    vector<int> bad = kahn(H, false);
    cout << "  vertices emitted: " << bad.size() << " of " << H.n << " -- ";
    for (int u : bad) cout << H.name[u] << " ";
    cout << "\n  stuck (in-degree never fell to 0):";
    vector<bool> out(H.n, false);
    for (int u : bad) out[u] = true;
    for (int u = 0; u < H.n; ++u) if (!out[u]) cout << " " << H.name[u];
    cout << "\n  verdict: " << ((int)bad.size() < H.n
             ? "cycle detected, no topological order exists" : "acyclic") << "\n";
    return 0;
}
```

Its output:

```
Kahn on the dressing DAG
  initial in-degrees: shirt=0 tie=1 jacket=2 belt=2 watch=0 undershorts=0 pants=1 shoes=3
socks=0
  step  1: emit shirt        freed: tie
  step  2: emit tie          freed: -
  step  3: emit watch        freed: -
  step  4: emit undershorts  freed: pants
  step  5: emit pants        freed: belt
  step  6: emit belt         freed: jacket
  step  7: emit jacket       freed: -
  step  8: emit socks        freed: shoes
  step  9: emit shoes        freed: -
  order: shirt tie watch undershorts pants belt jacket socks shoes
  valid topological order: yes

Kahn on the same graph plus the edge jacket -> undershorts
  vertices emitted: 4 of 9 -- shirt tie watch socks
  stuck (in-degree never fell to 0): jacket belt undershorts pants shoes
  verdict: cycle detected, no topological order exists
```

The in-degrees on the first line are the ones we counted by hand, the nine steps are **Table 11.4**, and the second run shows the cycle test working: four vertices escape, and the five that are stuck are the four-vertex cycle `undershorts` → `pants` → `belt` → `jacket` → `undershorts` plus `shoes`, which sits downstream of it and can never be freed.

11.3.14 Which of the two to use

They cost the same and both are correct, so the choice is made on other grounds.

Table 11.5 Choosing between the two topological-sort algorithms. Both are $\Theta(V + E)$ and both are correct; everything in this table is about engineering, not complexity.

Consideration	DFS finishing times	Kahn's in-degrees
Lines of code, given a DFS already written	2 extra	~12
Recursion	yes — deep graphs can overflow the stack	no, purely iterative
Cycle detection	yes, via a gray neighbor; reports the *edge*	yes, via $L< V$; reports the *stuck set*
Control over the output order	none: the search decides	full: choose the ready-set container
Detects a cycle **early**	yes, at the offending edge	no, only after the loop ends
Natural for an incremental / streaming setting	no	yes: feed new vertices into R as they become ready
Extends to longest-path / scheduling	awkward	directly (see §11.13.3)

In practice: if you already have a depth-first search and the graph is shallow, take the two-line version. If the graph is large and possibly deep, if you want the output in a particular order (alphabetical, or by priority), or if you want to *also* compute earliest start times as you go, take Kahn.

11.4 Spanning Trees

We leave directed graphs behind now. For the rest of this chapter every graph is **undirected**, **connected**, and **weighted**.

11.4.1 Definition

DEFINITION. A **tree** is a connected undirected graph with no cycles. A **spanning tree** of a connected graph $G = (V, E)$ is a subgraph T that (i) is a tree and (ii) contains **all** the vertices of G.

Both conditions are doing work. Dropping (i) and you could keep every edge, which is connected but has cycles. Dropping (ii) and a single edge would qualify, which is a tree but spans almost nothing. Together they say: *keep every place, throw away every redundant link, and keep everything reachable.*

Figure 11.4 illustrates the definition. On the left, a graph G on thirteen vertices with many more edges than it needs. On the right, one spanning tree of G: the same thirteen vertices, but only twelve edges, arranged so that the whole thing is still in one piece and there is exactly one route between any two vertices.

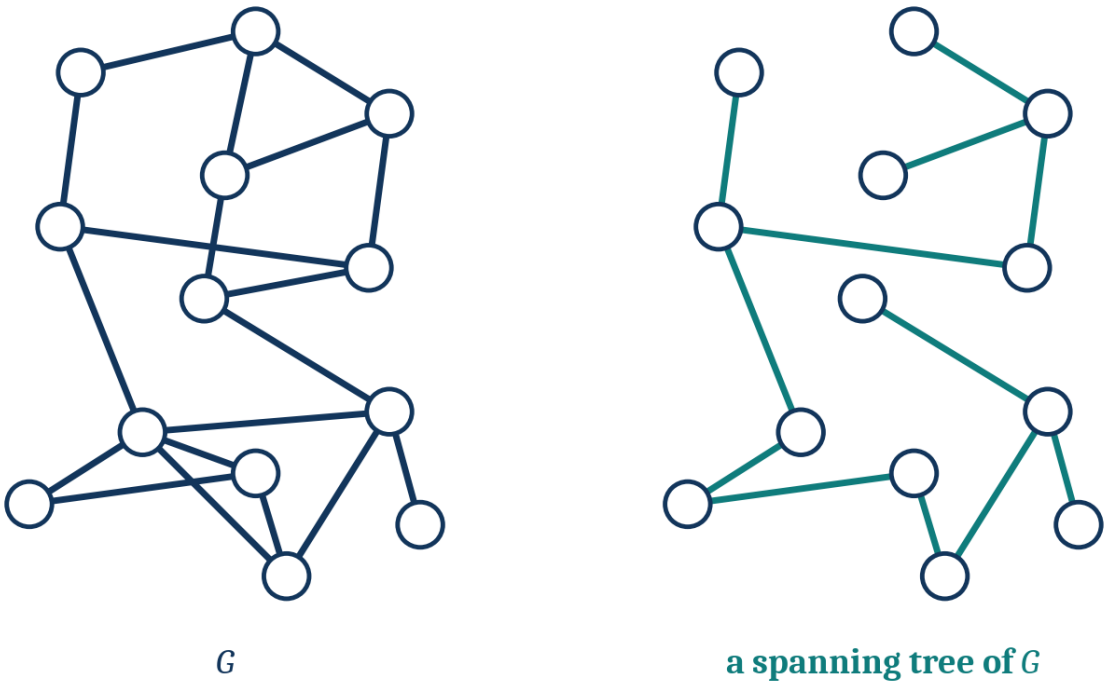


Figure 11.4 A spanning tree. **Left:** the graph G. **Right:** a spanning tree of G — every vertex of G is present, the graph is still connected, and no cycle remains. Count the edges on the right: twelve, for thirteen vertices.

IN PLAIN LANGUAGE. Take a road map of a province and rub out roads, one at a time, as long as you never cut a town off from the rest. Stop when rubbing out any further road would strand somebody. What is left is a spanning tree: every town still reachable, and not one road to spare.

11.4.2 A spanning tree has exactly $|V| - 1$ edges

This fact is used constantly — every algorithm in this chapter stops after adding exactly $|V| - 1$ edges — so it deserves a proof, not an assertion.

THEOREM 11.3. Let T be a tree on $n \geq 1$ vertices. Then T has exactly $n - 1$ edges.

Proof, by induction on n.

Base case, $n = 1$. A single vertex with no edges is connected and acyclic, so it is a tree, and it has $0 = 1 - 1$ edges. True.

Inductive step. Assume the theorem for all trees on fewer than n vertices, and let T be a tree on $n \geq 2$ vertices. We first claim T has a vertex of degree 1 — a **leaf**. Suppose not: every vertex has degree ≥ 2 (it cannot have degree 0, since T is connected and $n \geq 2$). Then start at any vertex and walk, each time leaving by an edge other than the one you arrived on; degree ≥ 2 guarantees such an edge always exists, so the walk never stops. Since V is finite, some vertex must be visited twice, and the portion of the walk between the two visits is a cycle. But T is acyclic. Contradiction, so a leaf ℓ exists.

Delete ℓ and its single incident edge. The result T' is still acyclic (deleting things cannot create a cycle) and still connected: any path in T between two surviving vertices could not have passed *through* ℓ, because entering ℓ and leaving again would require two distinct edges at ℓ, and ℓ has only one. So T' is a tree on $n-1$ vertices, and by the inductive hypothesis it has $n-2$ edges. Adding back ℓ and its edge gives T with $(n-2)+1 = n-1$ edges. ■

Two corollaries fall out, and both will be used:

- **A spanning tree of a connected graph on $|V|$ vertices has exactly $|V|-1$ edges.** Immediate, since a spanning tree is a tree on all $|V|$ vertices.
- **Adding any edge of G to a spanning tree creates exactly one cycle.** The tree already connects the two endpoints u and v by a unique path; the new edge $\{u, v\}$ closes that path into a cycle, and since the path was unique, the cycle is unique. This is the fact the proof of the cut property will lean on in §11.7.

A TRAP. "Exactly $n-1$ edges" does not by itself make a graph a tree. A graph on 4 vertices consisting of a triangle plus one isolated vertex has 3 edges and is not a tree — it has a cycle and it is disconnected. You need *two* of the three properties (connected, acyclic, $n-1$ edges) to get the third; any one alone is not enough.

11.4.3 How many spanning trees are there?

Enough to make brute force hopeless, which is the only fact we need. For the **complete** graph on n vertices, Cayley's formula [3] gives the count exactly: n^{n-2}. For $n = 9$, the size of the chapter's example graph, that is $9^7 = 4\,782\,969$ spanning trees — for a graph you can draw on a napkin. For $n = 20$ it is $20^{18} \approx 2.6 \times 10^{23}$. Enumerating them all and picking the cheapest is not an algorithm; it is a way of never finishing.

11.5 The Minimum Spanning Tree Problem

11.5.1 Weights

DEFINITION. A **weight function** on a graph $G = (V, E)$ is a map $w: E \to \mathbb{R}$ assigning a real number $w(u, v)$ to each edge $\{u, v\}$. The **weight of a subgraph** is the sum of the weights of its edges; in particular, for a spanning tree T,

$$w(T) = \sum_{\{u,v\} \in T} w(u, v). \tag{11.1}$$

So the setting is an undirected connected graph $G = (V, E)$ together with a weight function assigning a cost, a length or some other value to each edge, and the weight of a tree is the sum Equation (11.1). The word *weight* is deliberately neutral; in an application it is meters of cable, rupiah of construction cost, milliseconds of latency, or kilometers of road.

DEFINITION. A **minimum spanning tree** (MST) of a weighted connected graph G is a spanning tree T of G such that, with weights measured by Equation (11.1), $w(T) \leq w(T')$ for every spanning tree T' of G. The **minimum-spanning-tree problem** is: given G and w, find one.

Note the indefinite article: *a* minimum spanning tree. There may be several of equal weight, and §11.12 is about exactly when that happens.

11.5.2 A real problem with costs

Here is the problem in the shape it usually arrives. A cooperative of nine villages in East Java wants piped water. A survey has established, for each pair of villages close enough to be joined, how many hundreds of meters of pipe the connection would need — the terrain makes some short-looking hops expensive and some long ones cheap. Every village must end up connected to the network; it does not matter *how* the water reaches a village, only that it does. Pipe costs money per meter. **Which connections should be built?**

Three observations turn that into the MST problem exactly.

1. *The answer must be connected and span all nine villages* — that is the requirement.
2. *The answer must contain no cycle.* If it did, some pipe in that cycle could be removed and every village would still be connected — through the rest of the cycle — at strictly lower cost. So an optimal answer has no cycle. Hence it is a spanning tree.
3. *Among spanning trees we want the cheapest.* Which is the definition of an MST.

The same three lines apply, unchanged, to laying fiber between campus buildings, to choosing which flights an airline must keep to stay connected, to wiring a circuit board with the least solder, and — in §11.13 — to routing a laundry's shuttle vans.

A TRAP. The MST minimizes the **total** weight of the network. It does *not* minimize the distance between any particular pair of vertices. In **Figure 11.5** the MST route from one corner of the graph to the other may be far longer than the direct edge, because the direct edge was too expensive to keep. If you want cheap *journeys*, you want shortest paths, which is Chapter 13, and a shortest-path tree is generally **not** an MST. Confusing the two is the single most common error in this material.

11.5.3 An example graph

Figure 11.5 is a first weighted graph: eight vertices and ten edges, with weights 3, 5, 6, 7, 8, 9, 10, 12, 14 and 15. Look at it and ask the question. The edge of weight 15 sticks out into a corner — but it is the *only* edge reaching that corner vertex, so it cannot be dropped, expensive or not. The edge of weight 14, by contrast, is one of two ways of reaching its lower endpoint, and the other way, of weight 3, is far cheaper. The MST is the systematic version of that reasoning.

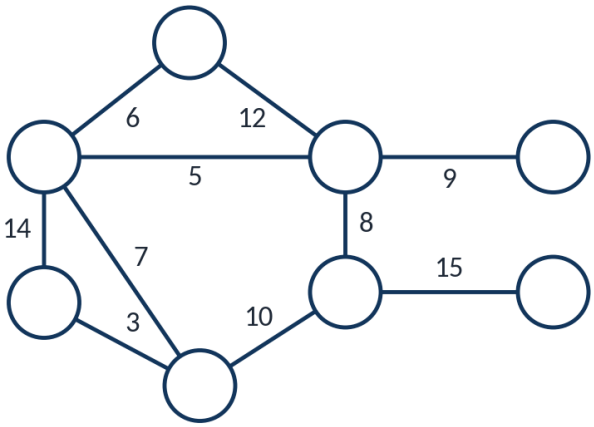


Figure 11.5 A weighted graph: an undirected connected graph with a weight on every edge. Some expensive edges are unavoidable (the 15 is the only way into its vertex); some cheap-looking ones are redundant. Distinguishing the two is the whole problem.

For the worked traces of Prim and Kruskal we use a slightly larger graph, on nine vertices a through i with fourteen edges. We will meet it in §11.9.5 and use it for everything from there on, so that the two algorithms can be compared on identical input.

11.5.4 Why brute force is hopeless, and what we need instead

By §11.4.3 there can be exponentially many spanning trees, so enumeration is out. Nor can we simply take the $|V| - 1$ cheapest edges: in **Figure 11.5** the three cheapest edges are 3, 5 and 6, and there is no guarantee they even form a forest, let alone a spanning one. (In the nine-vertex graph of §11.9.5 the edges (c, i) and (f, g) of weight 2, (c, f) of weight 4 and (g, i) of weight 6 form the cycle $c - i - g - f - c$, and all four are among the eight cheapest edges; taking the $|V| - 1$ cheapest edges *blindly* produces a cycle.)

What we need is a theorem that lets us commit to one edge at a time, permanently, with a guarantee that we have not ruled out the optimum. There are two such theorems, and the next two sections prove them. §11.6 proves that the problem has **optimal substructure**, so that a partial answer can be completed without looking back. §11.7 proves the **cut property**, which tells us which single edge is safe to commit to. Together they license a greedy algorithm — and, by Chapter 9's standard, licensing is exactly what greed requires.

11.6 Optimal Substructure of the MST

11.6.1 The claim

The claim is best stated with a picture. Let T be a minimum spanning tree of G, and let (u, v) be any edge of T. Remove that edge. Since T is a tree, removing one edge disconnects it into exactly two pieces — call them T_1 and T_2. (Exactly two: removing an edge from a tree cannot leave it connected, because the edge was the unique path between its endpoints; and it cannot break it into three, because each of the two remaining components is still connected internally.) Write V_1 and V_2 for their vertex sets, so $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \varnothing$.

Figure 11.6 draws the split, and **Figure 11.7** is the same drawing with the two halves circled and named.

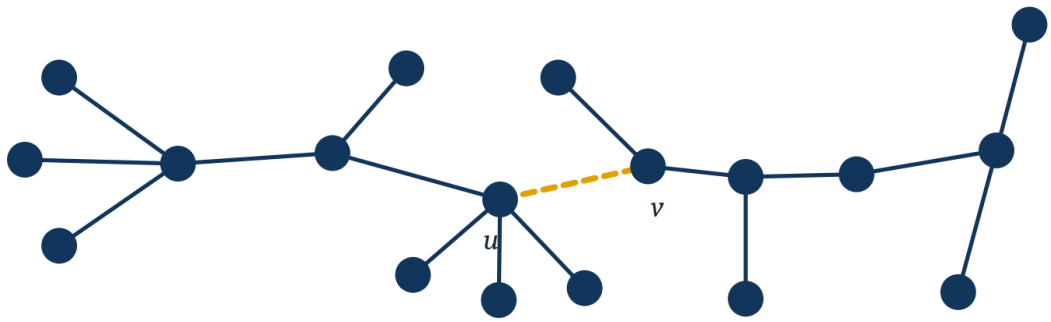


Figure 11.6 A minimum spanning tree T, with one of its edges (u, v) drawn as a dotted line. Removing that edge splits T into two pieces.

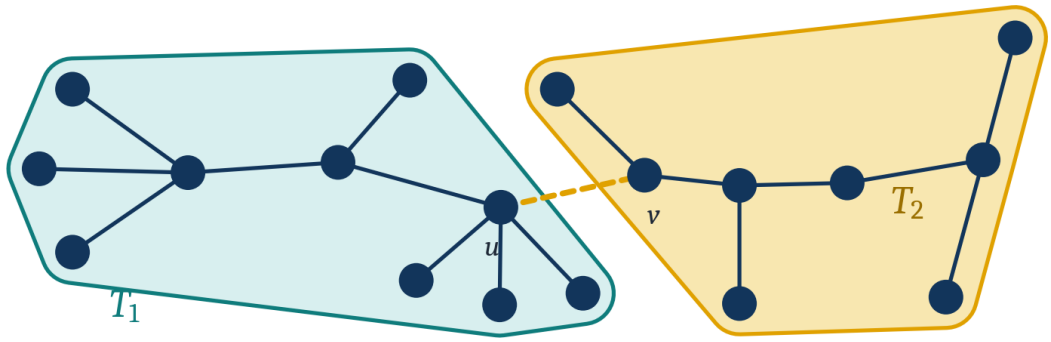


Figure 11.7 Figure 11.6 annotated: the two pieces left after removing the dotted edge (u, v) are named T_1 (left) and T_2 (right). The claim of Theorem 11.4 is that each of them is itself a minimum spanning tree, of the subgraph of G induced by its own vertices.

Because the edges of T are exactly those of T_1, those of T_2, and (u, v) itself, the weights add up:

$$w(T) = w(u, v) + w(T_1) + w(T_2). \tag{11.2}$$

Now let $G_1 = (V_1, E_1)$ be the subgraph of G **induced** by V_1 — meaning E_1 is every edge of G with both endpoints in V_1 — and define $G_2 = (V_2, E_2)$ the same way.

THEOREM 11.4 (optimal substructure). With the notation above, T_1 is a minimum spanning tree of G_1, and T_2 is a minimum spanning tree of G_2.

11.6.2 The proof

Proof. First, T_1 *is* a spanning tree of G_1: it is acyclic (a subgraph of a tree), it is connected (shown above), and its vertex set is exactly V_1. So the only question is whether it is minimum. By symmetry it is enough to argue about T_1.

Suppose it is not. Then there is a spanning tree $T_1{}'$ of G_1 with

$$w(T_1{}') < w(T_1). \tag{11.3}$$

Build a new subgraph of G by gluing the pieces back together with the cheaper half:

$$T' = T_1{}' \ \cup \ \{(u, v)\} \ \cup \ T_2.$$

Is T' a spanning tree of G? Its vertex set is $V_1 \cup V_2 = V$, so it spans. It is connected: $T_1{}'$ connects everything inside V_1, T_2 connects everything inside V_2, and the edge (u, v) — which has one endpoint in each, since it was the edge we removed — joins the two. And it is acyclic: it has

$$(|V_1| - 1) + 1 + (|V_2| - 1) = |V| - 1$$

edges by Theorem 11.3 applied to $T_1{}'$ and T_2, and a connected graph on $|V|$ vertices with $|V| - 1$ edges is a tree. So T' is a spanning tree of G.

What does it weigh? By the same decomposition as Equation (11.2), and then by the assumption Equation (11.3),

$$w(T') = w(u, v) + w(T_1{}') + w(T_2) \ < \ w(u, v) + w(T_1) + w(T_2) = w(T).$$

So T' is a spanning tree of G strictly cheaper than T. But T was a *minimum* spanning tree. Contradiction. Hence no such $T_1{}'$ exists, and T_1 is a minimum spanning tree of G_1. ■

> **IN PLAIN LANGUAGE.** This is Chapter 9's **cut-and-paste argument**, and it is worth naming because you will use it again and again. You assume some *part* of an optimal solution is not itself optimal; you cut that part out and paste in the better one; you check that the result is still a legal solution; and you observe that it is cheaper than something that was supposed to be cheapest. The contradiction proves the part was optimal all along.

11.6.3 What optimal substructure alone does not give us

Theorem 11.4 is necessary but not sufficient. It says that *once you know an edge of the MST*, the problem splits cleanly into two independent smaller problems that can be solved without ever looking back. What it conspicuously does not say is **which edge to take first**. Read it carefully: it begins "let (u, v) be any edge of T" — it assumes we already have an MST in hand.

Optimal substructure is the license to be greedy; it is not the greedy rule. For the rule we need a theorem of a different kind: one that names an edge, before any MST is known, and certifies that some MST contains it. That is the cut property, and it is the most important theorem in this chapter.

11.7 The Cut Property

This theorem is often called the **greedy-choice theorem** for minimum spanning trees. Everything that follows — Prim, Kruskal, Borůvka, and every MST algorithm anyone has published — is a different strategy for applying it. It deserves to be stated slowly.

11.7.1 Cuts, crossing edges, and respecting a set

Three pieces of vocabulary first.

> **DEFINITION.** A **cut** of $G = (V, E)$ is a partition of the vertices into two non-empty parts: a set $S \subseteq V$ and its complement $V - S$. We write the cut as $(S, V - S)$.
>
> An edge $\{u, v\}$ **crosses** the cut if exactly one of its endpoints is in S: that is, $u \in S$ and $v \in V - S$, or the other way round.
>
> A cut **respects** a set of edges A if no edge of A crosses it — every edge of A has both endpoints on the same side.

> **IN PLAIN LANGUAGE.** Draw a closed loop on the map. The villages inside are S, those outside are $V - S$. The pipes crossing your pencil line are the crossing edges. If you drew the line so that it cuts no pipe you have already committed to building, the cut *respects* your commitments.

Since G is connected and both sides are non-empty, at least one edge crosses every cut. So the phrase "the minimum-weight edge crossing the cut" always names something. If several crossing edges tie for minimum, it names any one of them.

DEFINITION. A minimum-weight edge crossing a cut is called a **light edge** for that cut.

11.7.2 The theorem

Here it is:

THEOREM 11.5 (the cut property, a.k.a. the greedy-choice property for MSTs). Let $G = (V, E)$ be a connected undirected weighted graph, let $S \subseteq V$ with $S \neq \varnothing$ and $S \neq V$, and let (u, v) be a minimum-weight edge of G connecting S to $V - S$. Then (u, v) belongs to **some** minimum spanning tree T of G.

Every qualifier matters, so read them off one by one.

- **"S is any non-empty proper subset."** You may draw the loop wherever you like. The theorem holds for all $2^{|V|} - 2$ cuts at once. This is what makes it usable: an algorithm gets to choose the cut that suits it, and two different choices give the two different algorithms of this chapter.
- **"minimum-weight edge connecting S to $V - S$."** Minimum *among the crossing edges only*. There may be far cheaper edges elsewhere in the graph; they are irrelevant to this cut.
- **"belongs to some minimum spanning tree."** Not *every* one. If two crossing edges tie for lightest, each of them belongs to some MST, and they may belong to different MSTs. The theorem promises existence, not uniqueness, and §11.12 explains the difference.

11.7.3 The proof, in a picture

Figure 11.8 draws the argument, and the proof is easier to follow with the picture in front of you than without it. The two shaded regions are S (left) and $V - S$ (right). The black lines are the edges of a minimum spanning tree T. The red edge at the bottom is our light edge (u, v), which — for the sake of argument — is *not* in T. The dotted line is the path that T does provide from u to v, and (x, y), drawn in teal, is the first edge of that path that crosses from S to $V - S$.

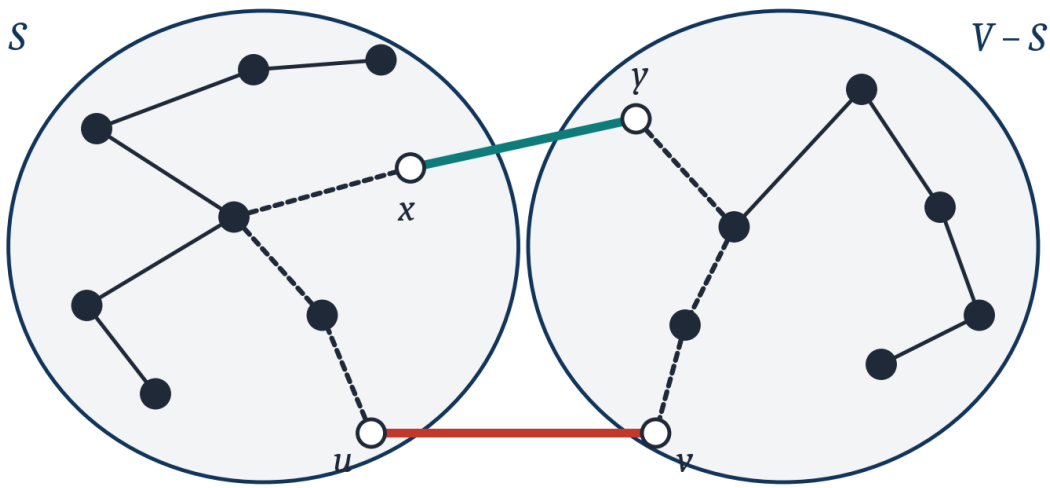


Figure 11.8 The exchange argument that proves Theorem 11.5. S and $V - S$ are the two blobs. The red edge (u, v) at the bottom is the light edge, assumed *not* to be in the tree. The dotted line is the unique path the tree provides from u to v; the teal edge (x, y) is the first edge of that path that crosses the cut. Swapping (x, y) out for (u, v) is the move the proof makes.

Proof. Let T be any minimum spanning tree of G. If $(u, v) \in T$ we are finished: (u, v) belongs to an MST, namely T. So suppose

$$(u, v) \notin T.$$

We will construct another spanning tree T' that does contain (u, v) and weighs no more than T. Since T is minimum, T' must then also be minimum, and the theorem follows.

Step 1: find the path. T is a spanning tree, so it connects every pair of vertices, and by §11.4.2 it does so by a **unique** path. Let P be the unique path in T from u to v — the dotted line in **Figure 11.8**.

Step 2: find the crossing edge. The path P starts at u, which is in S, and ends at v, which is in $V - S$. Walk along P from u. You begin inside S and you end outside it, so at some step you must step across the boundary. Let (x, y) be the **first** edge of P that crosses the cut, with $x \in S$ and $y \in V - S$. It exists, by that walking argument, and it is an edge of T since $P \subseteq T$.

Step 3: compare the weights. Here is the only place the hypothesis is used. The edge (x, y) crosses the cut $(S, V - S)$. The edge (u, v) is a **minimum-weight** edge crossing that cut. Therefore

$$w(u, v) \leq w(x, y). \tag{11.4}$$

Step 4: make the exchange. Define

$$T' = (T \setminus \{(x, y)\}) \cup \{(u, v)\}.$$

Claim: T' is a spanning tree. Removing (x, y) from T splits it into two components; one contains x, the other contains y. Now, the path P ran from u through x then y and on to v, so u is in the same component as x (the portion of P from u to x survives the removal, since (x, y) was the only edge of P we deleted) and v is in the same component as y. Hence adding the edge (u, v) rejoins the two components. The result is connected, spans V, and has $|V| - 1$ edges — one removed, one added — so by Theorem 11.3's corollary it is a tree. So T' is a spanning tree of G, and by construction $(u, v) \in T'$.

Step 5: price it. Using Equation (11.4) to replace $w(u, v)$ by the larger (or equal) $w(x, y)$:

$$w(T') = w(T) - w(x, y) + w(u, v) \;\leq\; w(T) - w(x, y) + w(x, y) = w(T).$$

So $w(T') \leq w(T)$. But T is a minimum spanning tree, so no spanning tree weighs less than T; hence $w(T') = w(T)$, and T' is itself a minimum spanning tree. It contains (u, v). ■

A shorter version of Step 5 says "this improves T — contradiction, since T was supposed to be an MST". That phrasing is aimed at the strict case: if $w(u, v) < w(x, y)$ then $w(T') < w(T)$ strictly, which contradicts the minimality of T outright. The version above handles ties as well, which is why it concludes with "belongs to *some* MST" rather than "belongs to *every* MST". Both readings are correct; the careful one is more useful, because real graphs have ties.

11.7.4 Reading it once more, slowly

The proof is short but it is doing something subtle, so here it is in plain words.

We are worried that committing to the cheapest crossing edge might be a mistake — that the true optimum uses some other crossing edge, and by grabbing ours we have ruined everything. The proof says: no. Suppose the optimum T avoids our edge (u, v). Then T still has to get from u to v

somehow, and whatever route it takes has to cross the boundary at least once. Find the place where it crosses. That crossing edge cannot be cheaper than ours — ours was the cheapest crossing edge there is. So take ours instead of theirs. The tree stays a tree, because we cut it in one place and rejoined it in exactly the place we cut. And the bill does not go up. Therefore an optimal tree containing our edge exists, and grabbing it was safe.

> **THE ONE-SENTENCE VERSION.** *The cheapest way across a boundary is always part of some cheapest network, because any other network has to cross that boundary somewhere, and you can always swap its crossing for yours without breaking it or paying more.*

11.7.5 The corollary the algorithms actually use

Theorem 11.5 talks about an empty slate. Both algorithms in this chapter have already committed to some edges and need the theorem to keep applying. Here is the version they use.

> **COROLLARY 11.6.** Let A be a subset of the edges of some minimum spanning tree of G. Let $(S, V - S)$ be any cut that **respects** A, and let (u, v) be a light edge for that cut. Then $A \cup \{(u, v)\}$ is also a subset of the edges of some minimum spanning tree of G.

Proof sketch. Let $T \supseteq A$ be an MST. If $(u, v) \in T$, done. Otherwise run the exchange of §11.7.3 to get $T' = (T \setminus \{(x, y)\}) \cup \{(u, v)\}$, which is an MST containing (u, v). We must check $A \subseteq T'$, that is, that the edge we threw away was not one of ours. But (x, y) crosses the cut, and the cut respects A, so no edge of A crosses it — hence $(x, y) \notin A$. Therefore $A \cup \{(u, v)\} \subseteq T'$. ■

That final line is exactly why the word *respects* was defined. It is the guarantee that the exchange never sacrifices an edge we have already promised to build.

11.7.6 A worked check

Take the nine-vertex graph of this chapter (drawn in **Figure 11.9** and used from §11.9 onwards) and let $S = \{a\}$. The edges crossing this cut are $\{a, b\}$ of weight 4 and $\{a, h\}$ of weight 8. The light edge is $\{a, b\}$, weight 4. Theorem 11.5 promises that some MST contains $\{a, b\}$ — and indeed both of that graph's two MSTs do (§11.12.3 finds them).

Now take $S = \{g, h\}$. The crossing edges are $\{a, h\}$=8, $\{b, h\}$=11, $\{h, i\}$=7, $\{g, i\}$=6, $\{f, g\}$=2. The light edge is $\{f, g\}$, weight 2 — and it is in both MSTs. Finally take $S = \{d\}$: crossing edges $\{c, d\}$=7, $\{d, e\}$=9, $\{d, f\}$=14; light edge $\{c, d\}$=7, again in both MSTs. Three different cuts, three safe commitments, no MST computed. That is the power of the theorem: it certifies edges one at a time, from purely local information.

11.8 The Generic MST Algorithm

11.8.1 Safe edges

Corollary 11.6 suggests a name for the kind of edge we are allowed to commit to.

> **DEFINITION.** Let A be a set of edges contained in some minimum spanning tree of G. An edge $(u, v) \notin A$ is **safe for** A if $A \cup \{(u, v)\}$ is still contained in some minimum spanning tree of G.

In a phrase, a *safe edge* is an edge that does not destroy A's property, and the property in question is precisely *being extendable to an MST*. Corollary 11.6 is then a machine for producing safe edges: **find a cut that respects A, and take a light edge for it.**

11.8.2 The generic algorithm

Which gives **Pseudocode 11.3**, the skeleton that both of this chapter's algorithms fill in:

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
```

Five lines, of which only line 3 is real work. Lines 1, 4 and 5 are bookkeeping, and line 2's test is just "have we got $|V| - 1$ edges yet?", by Theorem 11.3.

11.8.3 The loop invariant

> **INVARIANT.** *Before each iteration of the* `while` *loop, A is a subset of the edges of some minimum spanning tree of G.*

Initialization. Before the first iteration $A = \varnothing$, and the empty set is a subset of the edges of any MST, of which at least one exists (a connected finite graph has finitely many spanning trees, so one of minimum weight exists).

Maintenance. The body adds an edge that is safe for A, and "safe" is defined to be exactly the condition that keeps the invariant true.

Termination. The loop ends when A forms a spanning tree. At that point A is a spanning tree *and* a subset of the edges of some MST T. A spanning tree has $|V| - 1$ edges, and so does T; a set of $|V| - 1$ edges contained in a set of $|V| - 1$ edges is that set. So $A = T$, and A is a minimum spanning tree. ■

And the loop does terminate: each iteration adds exactly one edge, so after $|V| - 1$ iterations A has $|V| - 1$ edges. Does it also *form a spanning tree* by then? Yes — the invariant says A is contained in a spanning tree with $|V| - 1$ edges, so at that size it equals it. And line 3 can always find a safe edge while $|A| < |V| - 1$: take any cut that separates two components of A (such a cut exists, since A is then not spanning), and Corollary 11.6 supplies its light edge.

11.8.4 The more specific version

The skeleton is refined at once by substituting Corollary 11.6 for the word "safe", which gives **Pseudocode 11.4**:

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
```

This is now an algorithm with exactly one degree of freedom: **how do you choose the cut?** Line 3.1 is the only decision left, and it is the entire difference between the two algorithms that remain in this chapter.

11.8.5 Two ways to choose the cut

Table 11.6 The two algorithms of this chapter are the same algorithm with two different answers to line 3.1 of `MoreSpecific-MST`. Everything else — the correctness proof, the $|V| - 1$ edge count, the loop invariant — is shared.

	Prim–Jarník	Kruskal
What A looks like	a single tree, growing from a root r	a forest of several trees
The cut chosen	S = the vertices already in the tree	S = the vertices of the component containing one endpoint of the next candidate edge
Does it respect A?	yes: every edge of A has both ends inside the tree	yes: every edge of A lies wholly inside one component
How the light edge is found	a priority queue over the vertices outside S	sort all edges once, then test each in turn
Driven by	vertices	edges

Table 11.6 is the thesis of the rest of the chapter. Read it again after §11.10 and it will say more.

11.9 Prim–Jarník

11.9.1 The idea: one growing cloud

The algorithm was found by Vojtěch Jarník in 1930 and independently by Robert Prim in 1957 [4], [5]; this book calls it Prim–Jarník, which is fair to both. (Otakar Borůvka had in fact solved the problem in 1926, for the electrification of Moravia; Graham and Hell [6] tell the tangled story of who found what, and when.) It can be described in four lines, and they are worth setting out because they contain the whole design:

Vertex based algorithm. Grows one tree T, one vertex at a time. A cloud covering the portion of T already computed. Label the vertices v outside the cloud with $key[v]$ — the minimum weight of an edge connecting v to a vertex in the cloud, $key[v] = \infty$ if no such edge exists.

So A is always a single tree. The cut is always the same cut: S = the cloud, $V - S$ = everything else. That cut respects A automatically, because every edge of A has both endpoints in the cloud. And the light edge for it is the cheapest edge from the cloud to the outside — which is what key is for.

11.9.2 The two arrays: key and π

> **DEFINITION.** For each vertex v not yet in the tree, $key[v]$ is the weight of the cheapest edge joining v to *any* vertex already in the tree, and $\pi[v]$ is the tree vertex at the other end of that edge. If no edge joins v to the tree yet, $key[v] = \infty$ and $\pi[v] = \text{NIL}$.

Two invariants make the algorithm work, and both are easy to see:

- **The minimum of key over all vertices outside the tree is the weight of the light edge for the cut.** Because every crossing edge is an edge from some outside vertex v to the tree, and $key[v]$ is the cheapest such edge at v; minimizing over v minimizes over all crossing edges.
- **The pair $(\pi[v], v)$ for the minimizing v *is* that light edge.** By the definition of π.

So "extract the vertex of smallest key" and "add the edge $(\pi[u], u)$" together perform one iteration of `MoreSpecific-MST`. That is the entire algorithm.

11.9.3 The algorithm

Pseudocode 11.5 gives the procedure.

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)
```

Lines 01–05 set up: everything is outside the tree, everything is unreachable (∞) except the chosen root r, which is given key 0 so that it is extracted first. Line 07 takes the cheapest outside vertex and puts it in. Lines 08–11 then do the **only** other thing that needs doing: now that u is inside, some outside vertices may have a cheaper route into the tree than before — namely, through u. Lines 09–11 update those, and only those. The test `v in Q` is essential: once v is inside, its key is history and must not be touched.

> **A TRAP.** The condition on line 09 is $w(u,v) < key[v]$, strictly less. Not $\leq$, and not a comparison against $key[u]$ or against any running total. Prim is not Dijkstra: there is no accumulated path length here. $key[v]$ is the weight of **one** edge. Writing `key[u] + w(u,v) < key[v]` turns Prim into Dijkstra and gives you a shortest-path tree, which is a different tree. Exercise 11.8 makes the difference concrete on a three-vertex graph.

Notice also that the algorithm never explicitly builds a set of edges. The MST is recorded in π: when the loop finishes, the tree is $\{(\pi[v], v) : v \in V, v \neq r\}$, which has $|V| - 1$ edges by construction.

11.9.4 Priority queues

Line 07 needs the smallest key, and lines 10–11 change keys. The abstraction that supports both is a priority queue:

> **DEFINITION.** A **priority queue** maintains a set S of elements, each with an associated value called its **key**, and supports: • `BuildPQ(S)` — initialize the queue to contain the elements of S; • `ExtractMin(S)` — return and remove the element of S with the smallest key; • `ModifyKey(S, x, newkey)` — change the key of x (here always *decreasing* it, so it is often called `DecreaseKey`).

A **binary heap** — the structure of Chapter 5 — implements all three: `BuildPQ` in $O(n)$, and `ExtractMin` and `ModifyKey` in $O(\lg n)$ each. That is the standard choice, and the one the program below makes.

11.9.5 The example traced, step by step

The trace takes nine panels, reproduced here one per step. The graph has nine vertices $a \ldots i$ and fourteen edges; filled black vertices are in the tree, hollow ones are still in Q, and the gray highlight marks the edges committed so far. The root is a.

We will read the priority queue at every step as a list of `vertex:key/π` entries, sorted by key.

Step 1. Q holds all nine vertices with $key[a] = 0$ and every other key ∞. `ExtractMin` returns a. No edge is added — a is the root. Now a's two neighbors are updated: $key[b] \leftarrow 4$, $\pi[b] \leftarrow a$, and $key[h] \leftarrow 8, \pi[h] \leftarrow a$.

$$Q = \{\, b{:}\,4/a,\ \ h{:}\,8/a,\ \ c{:}\,\infty,\ \ d{:}\,\infty,\ \ e{:}\,\infty,\ \ f{:}\,\infty,\ \ g{:}\,\infty,\ \ i{:}\,\infty \,\}$$

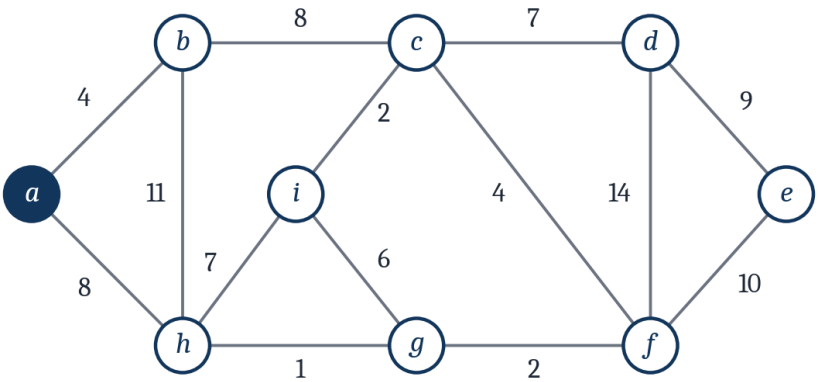


Figure 11.9 Step 1 of the Prim trace: the tree is just the root a. This is also the graph itself — nine vertices, fourteen edges, weights 1 to 14 — and it is the graph used for every trace in the rest of the chapter.

Step 2 (**Figure 11.10**). The smallest key in Q is $b{:}\,4$. Extract b, and commit the edge $(\pi[b], b) = (a, b)$ of weight 4. Running total 4. Now update from b: c has $key[c] = \infty > 8 = w(b, c)$, so $key[c] \leftarrow 8, \pi[c] \leftarrow b$; and h already has $key[h] = 8$, while $w(b, h) = 11$ is not smaller, so h is left alone.

$$Q = \{\, c{:}\,8/b,\ \ h{:}\,8/a,\ \ d{:}\,\infty,\ \ e{:}\,\infty,\ \ f{:}\,\infty,\ \ g{:}\,\infty,\ \ i{:}\,\infty \,\}$$

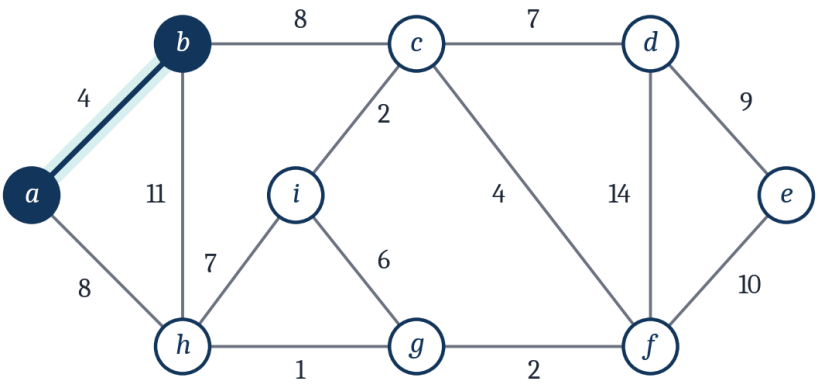


Figure 11.10 Step 2: the light edge for the cut $S = \{a\}$ is (a, b) of weight 4, so b joins the tree.

Step 3. Now there is a **tie**: c and h both have key 8. Either may be taken — Theorem 11.5 certifies both — and **Figure 11.11** takes c. Extract c, commit (b, c) of weight 8, total 12. Updating from c: $key[i] \leftarrow 2/c$, $key[f] \leftarrow 4/c$, $key[d] \leftarrow 7/c$.

$$Q = \{\, i{:}\,2/c,\ \ f{:}\,4/c,\ \ d{:}\,7/c,\ \ h{:}\,8/a,\ \ e{:}\,\infty,\ \ g{:}\,\infty \,\}$$

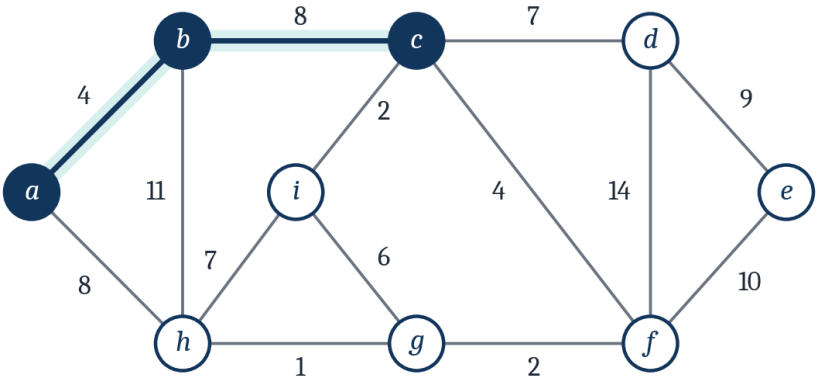


Figure 11.11 Step 3: with $S = \{a, b\}$ the crossing edges are $(b, c) = 8$, $(a, h) = 8$ and $(b, h) = 11$. Two of them tie for lightest; the trace takes (b, c).

Step 4 (Figure 11.12). Extract i: 2, commit (c, i) of weight 2, total 14. From i: $key[g] \leftarrow 6/i$ and $key[h]$ improves from $8/a$ to $7/i$, since $w(i, h) = 7 < 8$. Watch that last update closely — it is the algorithm changing its mind about *how* to reach h, without ever changing its mind about an edge already committed.

$$Q = \{\, f\!: 4/c,\ \ g\!: 6/i,\ \ d\!: 7/c,\ \ h\!: 7/i,\ \ e\!: \infty \,\}$$

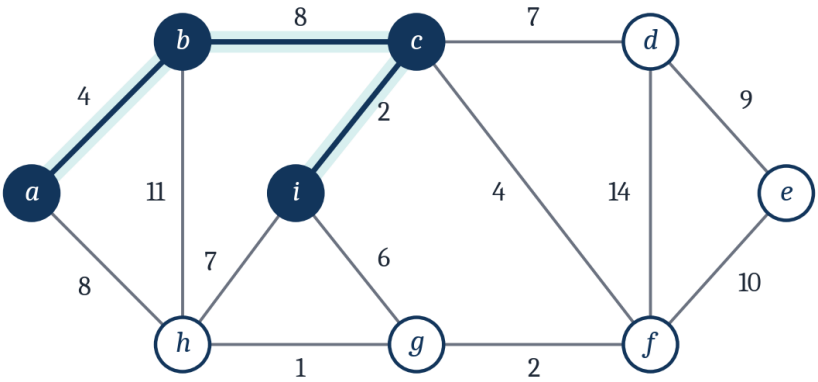


Figure 11.12 Step 4: (c, i) of weight 2 is the cheapest way out of $\{a, b, c\}$, so i joins. This is also the step at which h's key is *improved*, from 8 to 7.

Step 5 (Figure 11.13). Extract f: 4, commit (c, f) of weight 4, total 18. From f: $key[g]$ improves from $6/i$ to $2/f$, and $key[e] \leftarrow 10/f$. ($w(f, d) = 14$ is worse than $key[d] = 7$, so d is untouched.)

$$Q = \{\, g\!: 2/f,\ \ d\!: 7/c,\ \ h\!: 7/i,\ \ e\!: 10/f \,\}$$

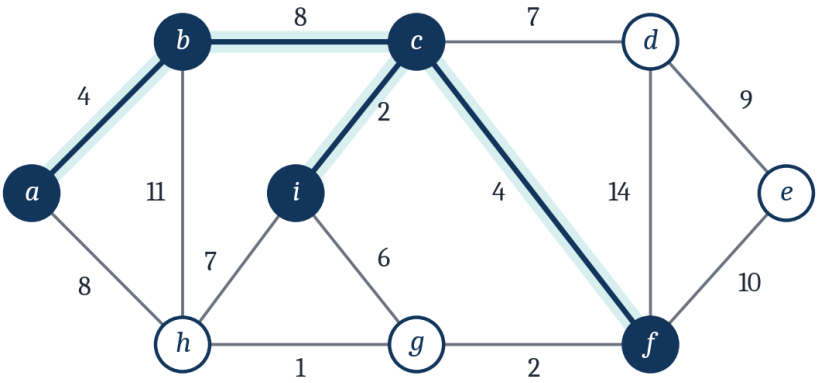


Figure 11.13 Step 5: (c, f) of weight 4 joins f. The tree is now $\{a, b, c, i, f\}$ and g's key drops from 6 to 2.

Step 6 (Figure 11.14). Extract g: 2, commit (f, g) of weight 2, total 20. From g: $key[h]$ improves again, from $7/i$ to $1/g$, because $w(g, h) = 1$.

$$Q = \{\, h\colon 1/g,\ \ d\colon 7/c,\ \ e\colon 10/f \,\}$$

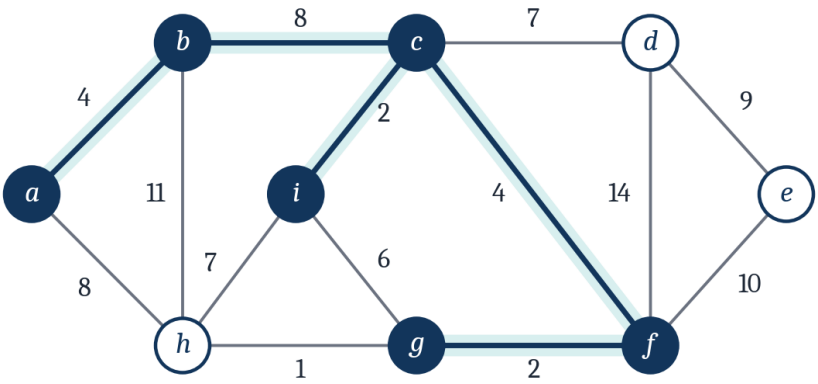


Figure 11.14 Step 6: (f, g) of weight 2 joins g. Vertex h has now had its key lowered three times — 8, then 7, then 1 — which is exactly what `DecreaseKey` is for.

Step 7 (Figure 11.15). Extract h: 1, commit (g, h) of weight 1, total 21. From h: its neighbors a, b, i are all already in the tree, so nothing changes.

$$Q = \{\, d\colon 7/c,\ \ e\colon 10/f \,\}$$

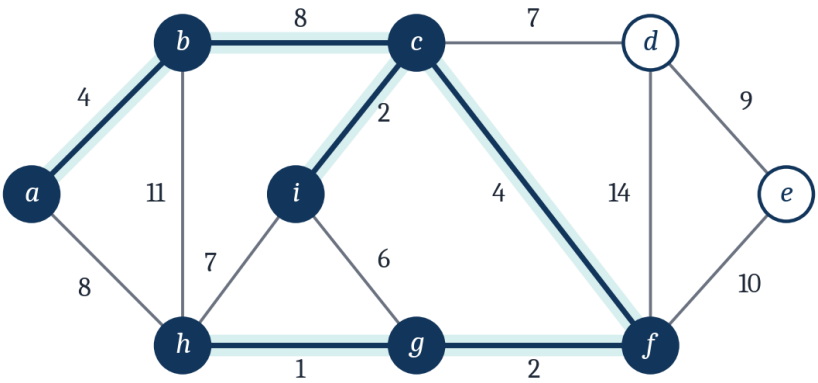


Figure 11.15 Step 7: the cheapest edge in the whole graph, (g, h) of weight 1, is only taken now — at the seventh step. Prim takes edges in the order the growing cloud reaches them, not in order of weight.

Step 8 (Figure 11.16). Extract d: 7, commit (c, d) of weight 7, total 28. From d: $key[e]$ improves from $10/f$ to $9/d$, since $w(d, e) = 9 < 10$.

$$Q = \{\, e\colon 9/d \,\}$$

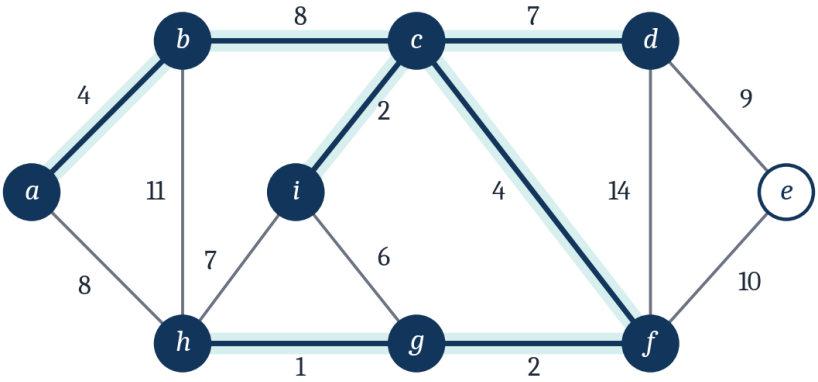


Figure 11.16 Step 8: (c, d) of weight 7 joins d, and e's key improves from 10 to 9.

Step 9. Extract e: 9, commit (d, e) of weight 9, total 37. Q is empty and the algorithm halts.

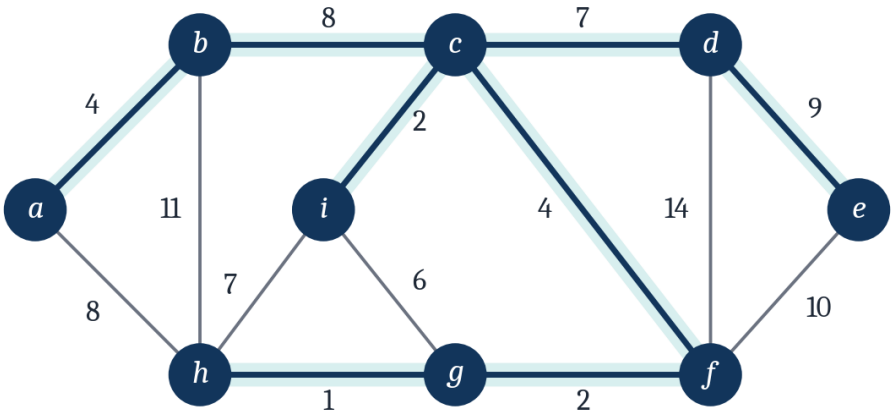


Figure 11.17 Step 9 — the final picture. All nine vertices are black and the eight highlighted edges form the minimum spanning tree: (a, b), (b, c), (c, i), (c, f), (f, g), (g, h), (c, d), (d, e), of total weight 37.

The whole run on one page:

Table 11.7 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)

WHAT TO NOTICE. The weights Prim commits to are 4, 8, 2, 4, 2, 1, 7, 9 — wildly out of order. Prim never looks at the graph as a whole; at each step it looks only at the boundary of its own cloud. The heaviest edge it takes (9) is taken last, and the lightest (1) seventh, purely because that is when the cloud reached them.

11.9.6 Why Prim is correct

Theorem. `MST-Prim` returns a minimum spanning tree.

Proof. We show Prim is `MoreSpecific-MST` with a particular choice of cut, and then quote §11.8.3.

Let A be the set of edges $\{(\pi[v], v) : v \notin Q, v \neq r\}$ — the edges committed so far — and let $S = V \setminus Q$ be the vertices already extracted. Three things to check:

1. *A is a tree spanning S.* By induction. Initially $S = \varnothing$ and $A = \varnothing$. When u is extracted, the edge added is $(\pi[u], u)$, and $\pi[u] \in S$ (the algorithm only ever sets $\pi[v] \leftarrow u$ at the moment u enters S), while u was not in S. So one new vertex and one new edge are attached to an existing tree: still a tree, now spanning $S \cup \{u\}$.
2. *The cut $(S, V - S)$ respects A.* By (1) every edge of A has both endpoints in S, so no edge of A crosses.
3. *The edge added is a light edge for that cut.* By the two invariants of §11.9.2: $key[v]$ is the cheapest edge from v to S, so $\min_{v \in Q} key[v]$ is the cheapest edge crossing the cut, and `ExtractMin` returns the v that attains it.

Those invariants themselves hold because lines 08–11 restore them immediately after every extraction: the only new crossing edges after u joins S are the edges out of u, and each is compared against the incumbent $key[v]$.

So every edge Prim adds is safe for A by Corollary 11.6, and by §11.8.3's termination argument the final A — which has $|V| - 1$ edges and is a spanning tree — is a minimum spanning tree. ■

11.9.7 The running time

Count the priority-queue operations, which is where all the cost is.

- `ExtractMin` (line 07) is called exactly once per vertex: $|V|$ times.
- `ModifyKey` (line 11) is called at most once per examination of an edge. Over the whole run, lines 08–09 examine each adjacency list once, and $\sum_u |Adj[u]| = 2|E|$ in an undirected graph. So `ModifyKey` runs $O(E)$ times.

Writing $T(\text{op})$ for the cost of one operation, the total is

$$\text{Time} \;=\; |V| \cdot T(\text{ExtractMin}) \;+\; O(E) \cdot T(\text{ModifyKey}). \tag{11.5}$$

With a **binary heap**, both operations cost $O(\lg V)$, so Equation (11.5) becomes

$$\text{Time} = O(V \lg V) + O(E \lg V) = O((V + E) \lg V) = O(E \lg V), \tag{11.6}$$

the last step because G is connected, so $|E| \geq |V| - 1$ and the E term dominates.

11.9.8 Three queues compared

The choice of priority queue changes the bound in Equation (11.5), as **Table 11.8** shows:

Table 11.8 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	$O(V\lg V + E)$

Reading the table as engineering advice:

- The **unsorted array** is best when the graph is **dense**. With $|E| = \Theta(V^2)$, $O(V^2)$ beats $O(E\lg V) = O(V^2\lg V)$. It is also the simplest thing to write, and for $|V|$ up to a few thousand it is often the fastest in wall-clock terms because it touches memory linearly.
- The **binary heap** is best for the ordinary sparse case, $|E| = O(V)$ to $O(V\lg V)$, and it is what `std::priority_queue` gives you for free.
- The **Fibonacci heap** achieves the asymptotically best bound, $O(V\lg V + E)$, and it does so by making the $O(E)$ `DecreaseKey` calls cost $O(1)$ each *amortized* — meaning that although an individual call may be slow, any sequence of m calls costs $O(m)$ in total. **That is precisely the kind of claim Chapter 14 exists to prove**, and the potential method developed there is how one proves it. In practice its constant factors are large enough that it rarely wins below graphs with millions of edges.

> **A TRAP.** The $O(E\lg V)$ and $O(V\lg V + E)$ bounds both assume the priority queue supports `DecreaseKey` on an arbitrary element — which needs a handle from each vertex into its position in the heap. `std::priority_queue` provides no such handle. The program below uses the standard workaround, explained in §11.9.10, and its cost is $O(E\lg E)$, not $O(E\lg V)$. Since $|E| \le |V|^2$, we have $\lg E \le 2\lg V$, so the two differ only by a constant factor — but it is worth knowing which one you have implemented.

11.9.9 The complete program

Listing 11.3 Prim–Jarník with a binary heap, tracing the queue at every step. Compile with `g++ -std=c++17 -O2 -o prim prim.cpp`.

```
// Prim-Jarnik with a binary heap, on the weighted graph of Section 11.9.5.
// Prints key[] and pi[] and the contents of Q at every step.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
using namespace std;

const int INF = INT_MAX;

struct Edge { int to, w; };

struct Graph {
    int n;
    vector<string>       name;
    vector<vector<Edge>> adj;
    explicit Graph(const vector<string>& names)
        : n((int)names.size()), name(names), adj(names.size()) {}
    void addEdge(int u, int v, int w) {            // undirected: store both ways
        adj[u].push_back({v, w});
        adj[v].push_back({u, w});
    }
};

// An entry in the heap. `key` first so that the default comparison on pair
// orders by key; `greater<>` turns std::priority_queue into a min-heap.
using Item = pair<int, int>;                       // (key, vertex)

int primJarnik(const Graph& G, int r, vector<int>& pi, vector<int>& added, bool trace) {
    vector<int>  key(G.n, INF);
    vector<char> inQ(G.n, 1);                      // inQ[v] == 1 means v is still in Q
    pi.assign(G.n, -1);
    key[r] = 0;

    priority_queue<Item, vector<Item>, greater<Item>> Q;
    for (int v = 0; v < G.n; ++v) Q.push({key[v], v});
```

Listing 11.3 (Continued) *Prim–Jarník with a binary heap, tracing the queue at every step. Compile with g++ -std=c++17 -O2 -o prim prim.cpp.*

```
    int total = 0, step = 0;
    while (!Q.empty()) {
        Item top = Q.top(); Q.pop();
        int u = top.second;
        if (!inQ[u] || top.first != key[u]) continue;   // a stale copy; skip it
        inQ[u] = 0;                                     // u joins the tree
        if (pi[u] != -1) { total += key[u]; added.push_back(u); }

        if (trace) {
            cout << "step " << setw(2) << ++step << ": extract " << G.name[u]
                 << "  key=" << (key[u] == INF ? string("INF") : to_string(key[u]));
            if (pi[u] != -1) cout << "  add edge (" << G.name[pi[u]] << "," << G.name[u]
                                  << ") w=" << key[u] << "  running total=" << total;
            cout << "\n";
        }

        for (const Edge& e : G.adj[u])
            if (inQ[e.to] && e.w < key[e.to]) {
                key[e.to] = e.w;
                pi[e.to]  = u;
                Q.push({key[e.to], e.to});         // lazy DecreaseKey: push a better copy
            }

        if (trace) {                               // print Q as (vertex,key,pi)
            vector<int> rest;
            for (int v = 0; v < G.n; ++v) if (inQ[v]) rest.push_back(v);
            sort(rest.begin(), rest.end(),
                 [&](int a, int b) { return key[a] != key[b] ? key[a] < key[b] : a < b; });
            cout << "        Q = {";
            for (size_t i = 0; i < rest.size(); ++i) {
                int v = rest[i];
                cout << (i ? ", " : "") << G.name[v] << ":"
                     << (key[v] == INF ? string("INF") : to_string(key[v]));
                if (pi[v] != -1) cout << "/" << G.name[pi[v]];
            }
            cout << "}\n";
        }
    }
    return total;
}
```

Listing 11.3 (Continued) *Prim–Jarník with a binary heap, tracing the queue at every step. Compile with g++ -std=c++17 -O2 -o prim prim.cpp.*

```
int main() {
    vector<string> names = { "a","b","c","d","e","f","g","h","i" };
    enum Vertex { Va, Vb, Vc, Vd, Ve, Vf, Vg, Vh, Vi };
    Graph G(names);
    G.addEdge(Va,Vb,4);  G.addEdge(Va,Vh,8);  G.addEdge(Vb,Vc,8);
    G.addEdge(Vb,Vh,11); G.addEdge(Vc,Vd,7);  G.addEdge(Vc,Vf,4);
    G.addEdge(Vc,Vi,2);  G.addEdge(Vd,Ve,9);  G.addEdge(Vd,Vf,14);
    G.addEdge(Ve,Vf,10); G.addEdge(Vf,Vg,2);  G.addEdge(Vg,Vh,1);
    G.addEdge(Vg,Vi,6);  G.addEdge(Vh,Vi,7);

    cout << "V = " << G.n << ", E = 14, root = a\n\n";
    vector<int> pi, added;
    int total = primJarnik(G, Va, pi, added, true);

    cout << "\nMST edges, in the order Prim added them:\n";
    int check = 0, cnt = 0;
    for (int v : added) {
        int w = 0;
        for (const Edge& e : G.adj[v]) if (e.to == pi[v]) w = e.w;
        cout << "  (" << G.name[pi[v]] << "," << G.name[v] << ") w=" << w << "\n";
        check += w; ++cnt;
    }
    cout << "  edge count = " << cnt << "  (|V|-1 = " << G.n - 1 << ")\n";
    cout << "  total weight = " << check << "   (accumulated during the run: "
         << total << ")\n";
    return 0;
}
```

Its output:

```
V = 9, E = 14, root = a

step  1: extract a  key=0
        Q = {b:4/a, h:8/a, c:INF, d:INF, e:INF, f:INF, g:INF, i:INF}
step  2: extract b  key=4  add edge (a,b) w=4  running total=4
        Q = {c:8/b, h:8/a, d:INF, e:INF, f:INF, g:INF, i:INF}
step  3: extract c  key=8  add edge (b,c) w=8  running total=12
        Q = {i:2/c, f:4/c, d:7/c, h:8/a, e:INF, g:INF}
step  4: extract i  key=2  add edge (c,i) w=2  running total=14
        Q = {f:4/c, g:6/i, d:7/c, h:7/i, e:INF}
step  5: extract f  key=4  add edge (c,f) w=4  running total=18
        Q = {g:2/f, d:7/c, h:7/i, e:10/f}
step  6: extract g  key=2  add edge (f,g) w=2  running total=20
        Q = {h:1/g, d:7/c, e:10/f}
step  7: extract h  key=1  add edge (g,h) w=1  running total=21
        Q = {d:7/c, e:10/f}
step  8: extract d  key=7  add edge (c,d) w=7  running total=28
        Q = {e:9/d}
step  9: extract e  key=9  add edge (d,e) w=9  running total=37
        Q = {}

MST edges, in the order Prim added them:
  (a,b) w=4
  (b,c) w=8
  (c,i) w=2
  (c,f) w=4
  (f,g) w=2
  (g,h) w=1
  (c,d) w=7
  (d,e) w=9
  edge count = 8  (|V|-1 = 8)
  total weight = 37   (accumulated during the run: 37)
```

Compare that output line by line with **Table 11.7**: the extraction order, every key, every predecessor, every running total, and the final weight of 37 all agree with the hand trace and with **Figure 11.17**. Note also that the tie at step 3 was broken in favor of c over h — `std::priority_queue` with `greater<pair<int,int>>` compares the second component when the first is equal, and c has the smaller index. That is the same choice **Figure 11.11** makes, which is why the program's trace and the figures line up.

11.9.10 A note on the lazy `DecreaseKey`

`std::priority_queue` cannot modify an element in place, so the program does not. Instead, when a key improves, it **pushes a second copy** of the vertex with the better key, and relies on the line

```
if (!inQ[u] || top.first != key[u]) continue;   // a stale copy; skip it
```

to throw away the obsolete copies when they surface. This is the standard *lazy deletion* pattern, and it is worth understanding rather than memorizing.

- It is **correct** because the best copy of a vertex always has the smallest key among its copies, so it surfaces first; at that moment the vertex is still in Q and its `top.first` equals the current `key[u]`. Every later copy either finds `inQ[u] == 0` (the vertex has already joined) or a mismatched key, and is discarded.
- It costs **space**: the queue may hold up to $O(E)$ entries rather than $O(V)$.
- It costs **time**: each push and pop is $O(\lg E)$ rather than $O(\lg V)$, and there are $O(E)$ of them, giving $O(E \lg E)$. As noted in §11.9.8, that is within a constant factor of $O(E \lg V)$.

The alternative — an indexed binary heap that stores, for each vertex, where in the heap array it currently sits — gives the textbook $O(E \lg V)$ of Equation (11.6) exactly and uses $O(V)$ space. Exercise 11.13 asks you to write it.

11.10 Kruskal

11.10.1 The idea: a forest that merges

Joseph Kruskal published his algorithm in 1956 [7], a year before Prim. It can be described in three lines:

Edge based algorithm. Add the edges one at a time, in increasing weight order. The algorithm maintains A — a forest of trees. An edge is accepted if it connects vertices of distinct trees.

The contrast with Prim is total. Prim keeps one tree and asks *which vertex next?* Kruskal keeps many trees and asks *is this edge useful?* It sorts every edge in the graph once, then walks the sorted list from cheapest to dearest, accepting an edge if and only if its two endpoints are currently in different trees. If they are in the same tree, the edge would close a cycle, so it is discarded.

> **IN PLAIN LANGUAGE.** Lay all the candidate roads on a table, cheapest at the top. Take them one at a time. If a road joins two villages that already have a route between them, throw it away — it is redundant. Otherwise build it. When you have built $|V| - 1$ roads, stop: everything is connected and nothing is wasted.

The correctness follows from Corollary 11.6, as we will show in §11.10.11. But first: the algorithm needs to answer, cheaply, the question *are these two vertices in the same tree?* And it needs to answer it $|E|$ times. That data-structure problem is the whole implementation difficulty, and it is worth doing properly.

11.10.2 Disjoint sets, from scratch

We need to maintain a **partition** of V: a collection of pairwise disjoint sets whose union is V. Initially each vertex is alone; as edges are accepted, sets merge. Three operations are needed:

> **DEFINITION.** A **disjoint-set** (or **union–find**) structure maintains a partition $S = \{S_1, S_2, \dots\}$ and supports: • MakeSet(S, x): $S \leftarrow S \cup \{\{x\}\}$ — add a new set containing only x; • Union(S_i, S_j): $S \leftarrow S - \{S_i, S_j\} \cup \{S_i \cup S_j\}$ — replace two sets by their union; • FindSet(S, x): return the unique $S_i \in S$ with $x \in S_i$.

In practice `FindSet` returns not the set but a **representative** of it — one designated member, the same one for every element of the set. Then "are u and v in the same set?" becomes `FindSet(u) == FindSet(v)`, a comparison of two integers.

11.10.3 Attempt 1: sets as lists

The first implementation, and a good one to see before the better one, represents each set as a **linked list**, with every element pointing back at the first element, which serves as the representative. **Figure 11.18** draws it.

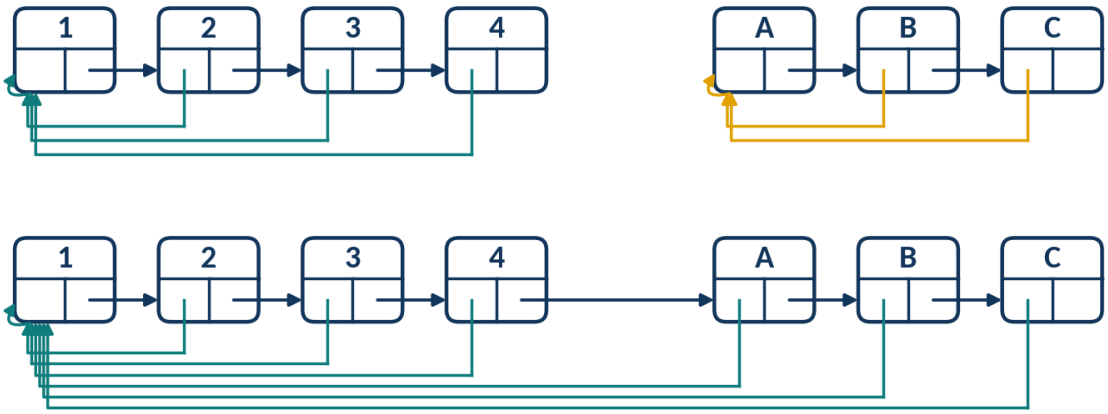


Figure 11.18 Disjoint sets as lists. **Top:** two sets, $\{1,2,3,4\}$ and $\{A, B, C\}$; within each, every element carries a back-pointer to the first element, which is the representative. **Bottom:** the result of uniting them — the shorter list has been appended and every one of its elements re-pointed at the new representative, 1.

The costs are easy to read off the picture:

- `FindSet(x)` is $O(1)$: follow one pointer.
- `Union(u,v)` must re-point every element of one list at the other list's head. If we always move the **smaller** list into the larger — "add a smaller list to a larger one" — the cost is $O(\min\{|C(u)|, |C(v)|\})$, where $C(x)$ is the set containing x.

A beautiful counting argument then bounds the total cost of all the unions. Let $t(v)$ be the number of times vertex v is moved to a new list. **Each time v is moved, the size of the set containing v at least doubles** — because v was in the smaller of the two sets, so the set it lands in is at least twice the size of the one it left. A set can double at most $\lg|V|$ times before it contains everything, so

$$t(v) \le \lg|V| \quad \text{for every } v,$$

and the total work done by all the unions in the whole run is

$$\sum_{v \in V} t(v) \;\le\; |V|\lg|V| \;=\; O(V\lg V). \tag{11.7}$$

The bound Equation (11.7) is already good enough for Kruskal, as §11.10.12 will show. But there is a better structure, and it is the one everybody uses.

11.10.4 Attempt 2: sets as trees

Represent each set as a **tree**, stored as an array `parent[]`. Each element points at its parent; the root points at itself and is the representative.

- `MakeSet(x)`: `parent[x] = x`. $O(1)$.
- `FindSet(x)`: walk up from x to the root. Cost = depth of x.
- `Union(x,y)`: find both roots and make one point at the other. $O(1)$ after the two finds.

Sedgewick and Wayne [8] call this the *quick-union* representation, and it is worth seeing in its broken form before it is repaired. `Union` is now trivially cheap, but `FindSet` costs the depth of the tree — and a naïve implementation can build a tree of depth $n-1$: unite 1 into 0, then 2 into 1, then 3 into 2, and you have a path, on which every find costs $\Theta(n)$. Two independent tricks fix this, and together they fix it spectacularly.

11.10.5 Trick 1: union by rank

Keep, for each root, an integer `rank` — an upper bound on the height of its tree. When uniting, **hang the tree of smaller rank under the root of larger rank.** If the ranks are equal, hang either under the other and increment the survivor's rank by one. **Pseudocode 11.6** spells it out.

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)
  parent[x] = x
  rank[x] = 0

UNION(x, y)
  rx = FIND-SET(x)
  ry = FIND-SET(y)
  if rx == ry
      return                          // already together
  if rank[rx] < rank[ry]
      swap rx and ry
  parent[ry] = rx
  if rank[rx] == rank[ry]
      rank[rx] = rank[rx] + 1
```

LEMMA 11.7. With union by rank, a tree whose root has rank k contains at least 2^k elements. Consequently every rank is at most $\lfloor \lg n \rfloor$, and every tree has height at most $\lfloor \lg n \rfloor$, so `FindSet` costs $O(\lg n)$.

Proof, by induction on k. A rank-0 root is a tree of one element, and $2^0 = 1$. A root reaches rank $k \ge 1$ only at a moment when two rank-$(k-1)$ roots are united; by induction each of those trees had at least 2^{k-1} elements, so the union has at least $2^{k-1} + 2^{k-1} = 2^k$. Since a tree can contain at most n elements, $2^k \le n$, so $k \le \lg n$. ■

Already this turns the worst case from $\Theta(n)$ per find into $O(\lg n)$. Note *why*: hanging the short tree under the tall one means the tall tree's height does not grow at all, and the only way to grow

height is to join two trees of exactly equal rank — which, by the lemma, requires doubling the population.

11.10.6 Trick 2: path compression

The second trick costs nothing and is almost free to write. **Whenever `FindSet(x)` walks from x up to the root, re-point every node it passed directly at the root.** **Pseudocode 11.7** does it.

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)
if parent[x] != x
    parent[x] = FIND-SET(parent[x])
return parent[x]
```

Three lines, and the recursion does all the work: the recursive call returns the root, and the assignment stores it. The next find on any node on that path costs one step.

> **IN PLAIN LANGUAGE.** The first person to ask "who is in charge here?" has to walk up the whole chain of command. On the way back down, they tell everybody they passed the answer directly. Nobody on that path ever has to walk again.

The trace in **Table 11.9** shows both tricks at work on eight elements. Watch the `rank` row to see union by rank, and watch `parent` change during the *finds* at the end to see path compression flatten the tree to depth 1.

Table 11.9 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 final 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
after `find(0)`…`find(7)`	**0 0 0 0 0 0 0 0**	3 0 1 0 2 0 1 0	1

11.10.7 Why the amortized cost is almost constant

With both tricks, the cost of a sequence of m operations on n elements is

$$O(m \cdot \alpha(n)), \tag{11.8}$$

where α is the **inverse Ackermann function**. The proof is due to Tarjan [9] — his monograph [10] develops the structure and its analysis at book length; it is not short, and it is exactly the sort of argument **Chapter 14** develops — you cannot bound a single `FindSet` by $\alpha(n)$, because one find genuinely can cost $\Theta(\lg n)$; you can only bound the *average over the whole sequence*, and averaging worst cases over a sequence is what amortized analysis means. Chapter 14's potential method is the tool.

What matters in Equation (11.8) is how absurdly slowly α grows, as **Table 11.10** shows. (The function A_k in the last row is the Ackermann-type function used to define α in [1]; all we need is its size.)

Table 11.10 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

WHAT THIS MEANS IN PRACTICE. $\alpha(n) \leq 4$ for every n you will ever meet. So a disjoint-set forest with union by rank and path compression performs each operation in *at most four times* the cost of a constant-time operation, amortized — and usually far less, because most finds land on an already-compressed node. You may treat `FindSet` and `Union` as $O(1)$ in engineering estimates, while knowing that the honest bound is $O(\alpha(n))$ and that it is not quite constant. (Fredman and Saks proved in 1989 that $\Omega(\alpha(n))$ amortized time per operation is unavoidable for *any* data structure solving this problem, even in the generous cell-probe model [11], so this is the end of the road.)

11.10.8 The disjoint-set program

Listing 11.4 is the structure on its own, printing the arrays after every operation so that **Table 11.9** can be checked line by line.

Listing 11.4 A disjoint-set forest with union by rank and path compression. Compile with `g++ -std=c++17 -O2 -o dsu dsu.cpp`.

```
// A disjoint-set forest with union by rank and path compression,
// printing the parent array after every operation so the structure is visible.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;

class DisjointSets {
public:
    explicit DisjointSets(int n) : parent(n), rank_(n, 0), sets(n) {
        for (int i = 0; i < n; ++i) parent[i] = i;   // MakeSet: every element alone
    }

    // FindSet with path compression: after the call, every node on the path
    // from x to the root points straight at the root.
    int findSet(int x) {
        if (parent[x] != x) parent[x] = findSet(parent[x]);
        return parent[x];
    }

    // Union by rank: the shorter tree is hung under the taller one, so the
    // height only grows when two trees of equal rank are joined.
    bool unite(int x, int y) {
        int rx = findSet(x), ry = findSet(y);
        if (rx == ry) return false;                  // already together
        if (rank_[rx] < rank_[ry]) swap(rx, ry);
        parent[ry] = rx;                             // ry's tree hangs under rx
        if (rank_[rx] == rank_[ry]) ++rank_[rx];     // equal ranks: height grows by 1
        --sets;
        return true;
    }

    int count() const { return sets; }
    int rankOf(int x) const { return rank_[x]; }
    const vector<int>& parents() const { return parent; }

private:
    vector<int> parent, rank_;
    int         sets;
};
```

Listing 11.4 (Continued) *A disjoint-set forest with union by rank and path compression. Compile with g++ -std=c++17 -O2 -o dsu dsu.cpp.*

```
void dump(const string& label, const DisjointSets& D, int n) {
    cout << left << setw(16) << label << right << " parent =";
    for (int i = 0; i < n; ++i) cout << setw(3) << D.parents()[i];
    cout << "   rank =";
    for (int i = 0; i < n; ++i) cout << setw(2) << D.rankOf(i);
    cout << "   sets=" << D.count() << "\n";
}

int main() {
    const int n = 8;
    DisjointSets D(n);
    cout << "elements 0..7, each in its own set\n";
    dump("after MakeSet", D, n);

    struct Op { int a, b; };
    vector<Op> ops = { {0,1}, {2,3}, {4,5}, {6,7}, {0,2}, {4,6}, {0,4}, {1,5} };
    for (const Op& o : ops) {
        bool merged = D.unite(o.a, o.b);
        dump("Union(" + to_string(o.a) + "," + to_string(o.b) + ")"
             + (merged ? "" : " noop"), D, n);
    }

    cout << "\nevery element now reports the same representative:\n  ";
    for (int i = 0; i < n; ++i) cout << "find(" << i << ")=" << D.findSet(i) << " ";
    cout << "\n";
    dump("after finds", D, n);

    cout << "\nwitness that path compression flattens: a chain built by hand\n";
    DisjointSets C(6);
    // Force a deep-ish tree by uniting in increasing order of rank.
    C.unite(0,1); C.unite(2,3); C.unite(0,2); C.unite(4,5); C.unite(0,4);
    dump("before find(5)", C, 6);
    cout << "find(5) = " << C.findSet(5) << "\n";
    dump("after find(5)", C, 6);
    return 0;
}
```

Its output:

```
elements 0..7, each in its own set
after MakeSet    parent =  0  1  2  3  4  5  6  7   rank = 0 0 0 0 0 0 0 0   sets=8
Union(0,1)       parent =  0  0  2  3  4  5  6  7   rank = 1 0 0 0 0 0 0 0   sets=7
Union(2,3)       parent =  0  0  2  2  4  5  6  7   rank = 1 0 1 0 0 0 0 0   sets=6
Union(4,5)       parent =  0  0  2  2  4  4  6  7   rank = 1 0 1 0 1 0 0 0   sets=5
Union(6,7)       parent =  0  0  2  2  4  4  6  6   rank = 1 0 1 0 1 0 1 0   sets=4
Union(0,2)       parent =  0  0  0  2  4  4  6  6   rank = 2 0 1 0 1 0 1 0   sets=3
Union(4,6)       parent =  0  0  0  2  4  4  4  6   rank = 2 0 1 0 2 0 1 0   sets=2
Union(0,4)       parent =  0  0  0  2  0  4  4  6   rank = 3 0 1 0 2 0 1 0   sets=1
Union(1,5) noop  parent =  0  0  0  2  0  0  4  6   rank = 3 0 1 0 2 0 1 0   sets=1

every element now reports the same representative:
  find(0)=0 find(1)=0 find(2)=0 find(3)=0 find(4)=0 find(5)=0 find(6)=0 find(7)=0
after finds      parent =  0  0  0  0  0  0  0  0   rank = 3 0 1 0 2 0 1 0   sets=1

witness that path compression flattens: a chain built by hand
before find(5)   parent =  0  0  0  2  0  4   rank = 2 0 1 0 1 0   sets=1
find(5) = 0
after find(5)    parent =  0  0  0  2  0  0   rank = 2 0 1 0 1 0   sets=1
```

The last three lines are path compression caught in the act: before the call, element 5 pointed at 4, which pointed at 0. After one `find(5)`, element 5 points straight at 0, and it will never walk that path again.

11.10.9 The algorithm

With the structure in hand, Kruskal's algorithm is the nine lines of **Pseudocode 11.8**:

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
```

Lines 02–03 start with $|V|$ singleton sets — a forest of $|V|$ isolated vertices, which is a legal starting forest. Line 04 sorts. Lines 05–08 walk the sorted edges; line 06 is the cycle test, and it is exactly "are these two vertices already connected?"

11.10.10 The example traced, edge by edge

The trace takes fourteen panels — one per edge of the graph. We take them two at a time. The graph is the same nine-vertex graph as **Figure 11.9**, so the two algorithms can be compared directly. The sorted edge list is

$$(g,h)1,\ (c,i)2,\ (f,g)2,\ (a,b)4,\ (c,f)4,\ (g,i)6,\ (c,d)7,\ (h,i)7,\ (a,h)8,\ (b$$

In each figure below an **arrow** marks the edge currently under consideration; if it becomes highlighted, it was accepted.

Edges 1 and 2. The forest is nine singletons: $\{a\},\{b\},\ldots,\{i\}$. Edge (g,h) of weight 1: `Find(g)`$=g \neq h=$`Find(h)`, different, so accept. Total 1. Now (c,i) of weight 2: `Find(c)`$=c \neq i$, accept. Total 3.

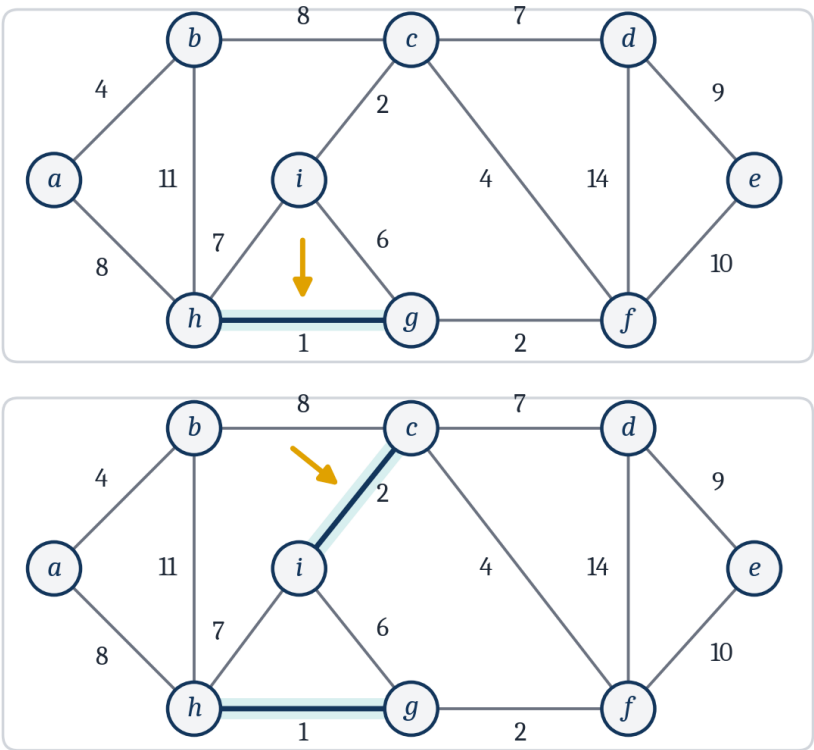


Figure 11.19 The first two Kruskal steps. **Top:** (g, h) of weight 1 — the cheapest edge in the graph — is accepted. **Bottom:** (c, i) of weight 2 is accepted. Unlike Prim, Kruskal starts with the globally cheapest edge, wherever it happens to be.

Edges 3 and 4 (Figure 11.20). Forest $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f\}, \{g, h\}$. Edge (f, g) of weight 2: `Find(f)`$=f$, `Find(g)`$=g$, different, accept — and now $\{f, g, h\}$ is one tree. Total 5. Edge (a, b) of weight 4: both singletons, accept. Total 9.

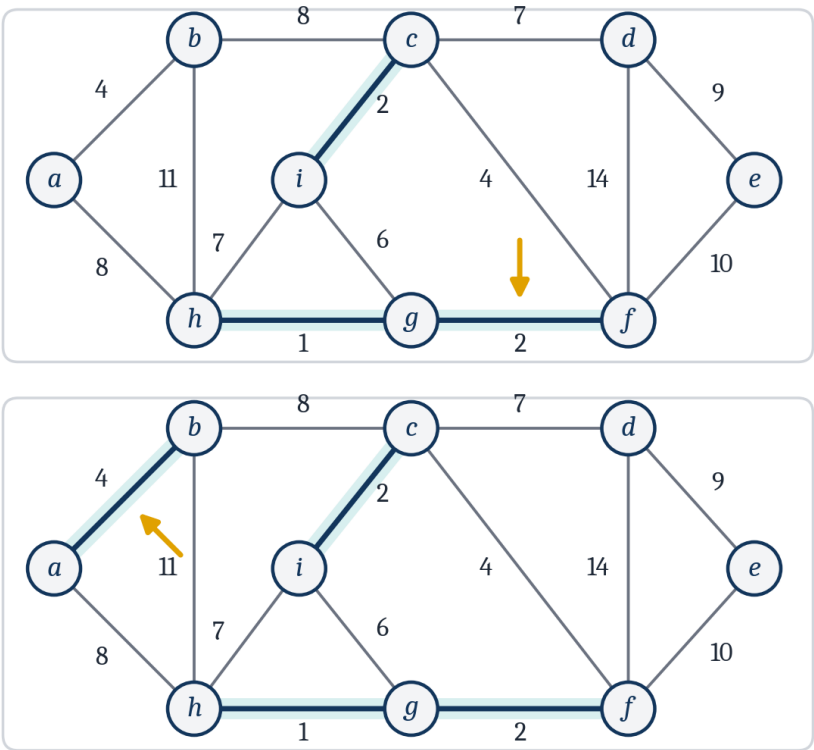


Figure 11.20 Top: (f, g) of weight 2 accepted, merging $\{f\}$ with $\{g, h\}$. **Bottom:** (a, b) of weight 4 accepted. Notice the forest now has four separate trees growing at once — something Prim never does.

Edges 5 and 6 (Figure 11.21). Forest $\{a, b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$. Edge (c, f) of weight 4: `Find(c)`$=c$, `Find(f)`$=g$ — different, accept, merging into $\{c, f, g, h, i\}$. Total 13. Then edge (g, i) of weight 6: `Find(g)`$=c$ and `Find(i)`$=c$ — **the same**. Rejected: g and i are already connected, by the route $g - f - c - i$.

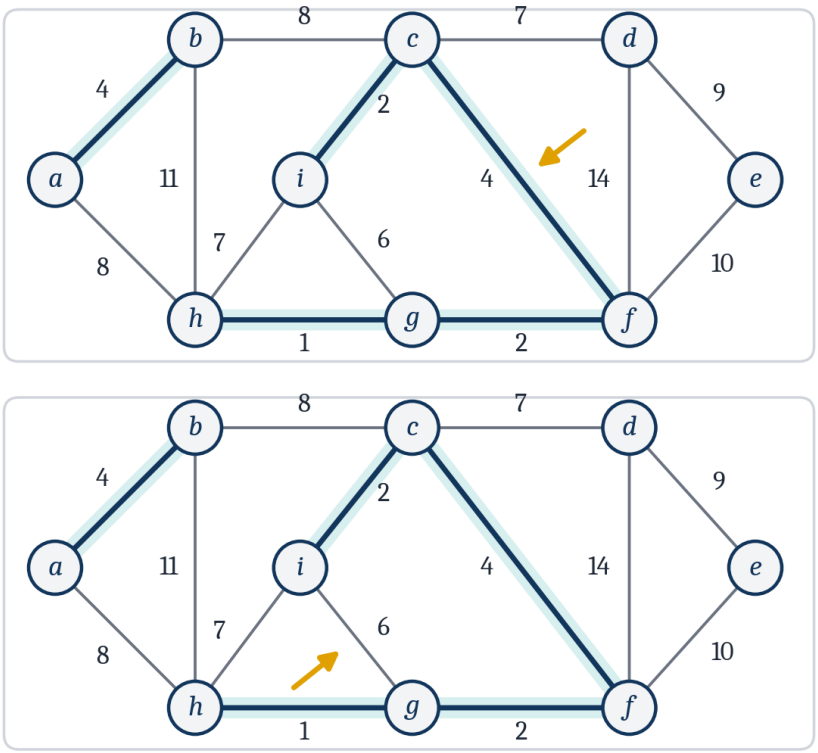


Figure 11.21 Top: (c, f) of weight 4 accepted, merging two trees into $\{c, f, g, h, i\}$. **Bottom:** the first rejection — (g, i) of weight 6 is *not* highlighted, because g and i are already in one tree and the edge would close a cycle.

Edges 7 and 8 (Figure 11.22). Forest $\{a, b\}, \{c, f, g, h, i\}, \{d\}, \{e\}$. Edge (c, d) of weight 7: different sets, accept; total 20. Edge (h, i) of weight 7: `Find(h)`$=$`Find(i)`$=c$, reject.

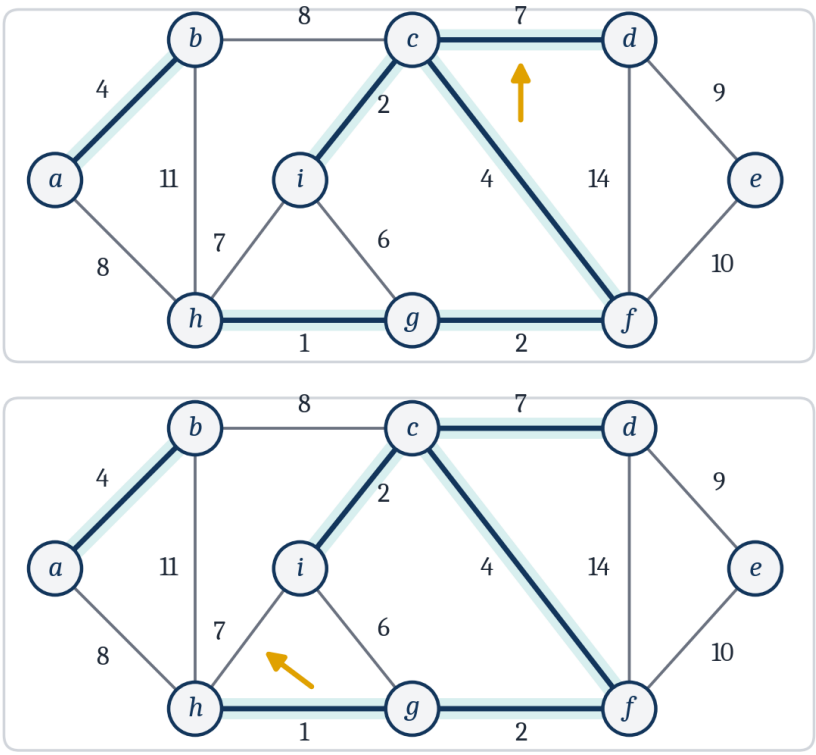


Figure 11.22 Top: (c, d) of weight 7 accepted. **Bottom:** (h, i), also of weight 7, rejected. Two edges of equal weight, opposite fates — the decision has nothing to do with the weight and everything to do with the partition.

Edges 9 and 10. Forest $\{a, b\}, \{c, d, f, g, h, i\}, \{e\}$. Edge (a, h) of weight 8: `Find(a)` $= a$, `Find(h)` $= c$, different, accept; total 28; the two big trees merge. Edge (b, c) of weight 8: now `Find(b)` $=$ `Find(c)` $= c$, reject.

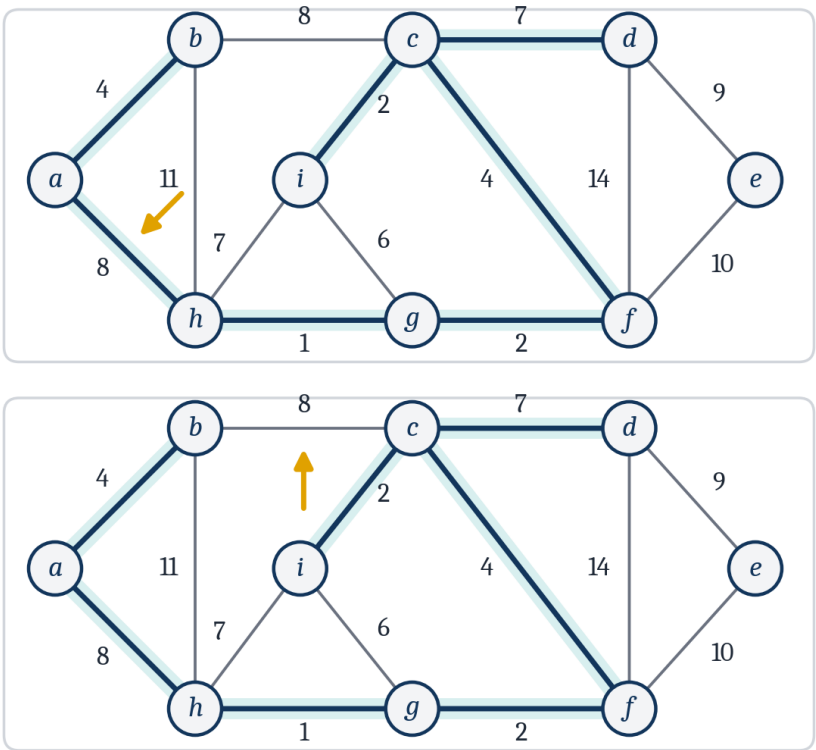


Figure 11.23 Top: (a, h) of weight 8 accepted. **Bottom:** (b, c) of weight 8 rejected. **This is the step at which Kruskal's answer diverges from Prim's:** Prim took (b, c) and never considered (a, h); Kruskal does the opposite. Both trees weigh 37 (see §11.12).

Edges 11 and 12 (Figure 11.24). Forest $\{a, b, c, d, f, g, h, i\}, \{e\}$. Edge (d, e) of weight 9: e is still alone, accept; total 37, and the forest becomes a single tree. Edge (e, f) of weight 10: same set now, reject.

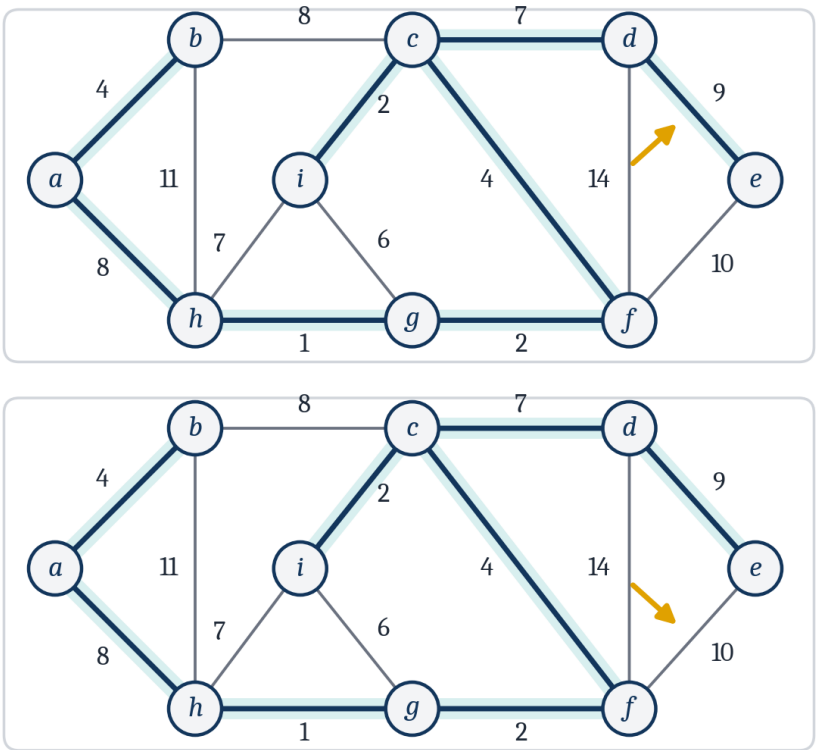


Figure 11.24 Top: (d, e) of weight 9 accepted — the eighth and last edge, bringing the isolated vertex e into the tree. **Bottom:** (e, f) of weight 10 rejected.

Edges 13 and 14. The forest is already a single tree of nine vertices, so every remaining edge must be rejected. (b, h) of weight 11: rejected. (d, f) of weight 14: rejected.

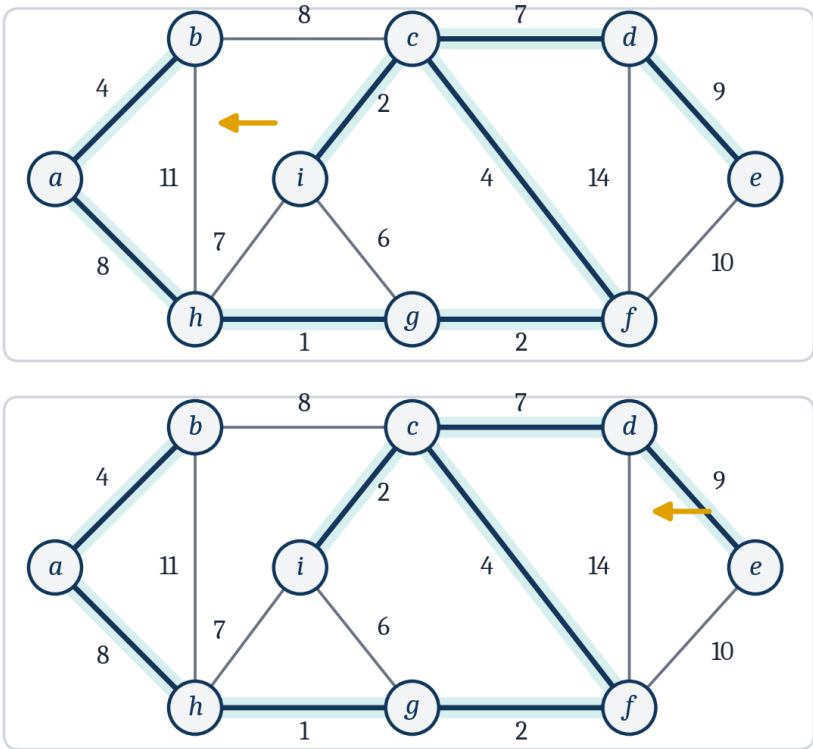


Figure 11.25 The last two panels: (b, h) of weight 11 and (d, f) of weight 14 are both rejected, and the highlighted eight edges are the finished minimum spanning tree of total weight 37 — the same weight as Prim's answer in **Figure 11.17**, by a different set of edges.

The run on one page:

Table 11.11 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
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

Eight accepts and six rejects — and $8 = |V| - 1$, as Theorem 11.3 requires. Note that every edge from #12 onward is rejected: once the forest is a single tree, nothing more can be added, so a practical implementation stops as soon as it has accepted $|V| - 1$ edges rather than scanning the tail of the list.

11.10.11 Why Kruskal is correct

Theorem. `MST-Kruskal` returns a minimum spanning tree.

Proof. Again we show it is `MoreSpecific-MST` in disguise. Let A be the set of edges accepted so far. Two observations:

A is always a forest. An edge is accepted only when its endpoints lie in different components of A, and adding an edge between two distinct components cannot create a cycle. So A is acyclic throughout, and its components are exactly the sets of the disjoint-set structure.

Every accepted edge is safe for A. Suppose edge (u, v) is accepted. Let C_u be the component of A containing u, and take the cut $S = C_u$, $V - S$ = everything else. Then: - The cut **respects** A: every edge of A lies wholly inside one component, and C_u is a whole component, so no edge of A crosses. - (u, v) **crosses** it, since $v \notin C_u$ (that is why the edge was accepted). - (u, v) is a **light** edge for it. This is the key step. Suppose some edge (x, y) crossing the cut had $w(x, y) < w(u, v)$. Then (x, y) appears strictly earlier in the sorted list, so Kruskal already examined it. What did it do? At the moment (x, y) was examined, either its endpoints were in different components — in which case it was accepted, and x and y have been in the same component of A ever since, contradicting the fact that (x, y) crosses the cut *now* (one endpoint in C_u, one outside) — or its endpoints were already in the same component, in which case they still are, and again (x, y) cannot cross the cut now. Either way, a contradiction. So no crossing edge is cheaper than (u, v); it is light.

By Corollary 11.6, $A \cup \{(u, v)\}$ is still contained in some MST. And when the scan ends, A has $|V| - 1$ edges: it cannot have fewer, because if two components remained, some edge of the connected graph G would join them, and that edge — examined at some point in the scan — would have been accepted. So A is a spanning tree contained in an MST, hence equal to it. ■

> **WHERE THE SORT EARNS ITS KEEP.** The whole light-edge argument above rests on one sentence: *any cheaper crossing edge would have been examined earlier.* That is true only because the edges are processed in non-decreasing weight order. Remove the sort and Kruskal is not merely slower — it is wrong.

11.10.12 The running time

Count the three phases:

- **Initialization.** $|V|$ calls to `MakeSet`, $O(1)$ each: $O(V)$.
- **Sorting.** $\Theta(E \lg E)$ with any comparison sort.
- **The scan.** $O(E)$ calls to `FindSet` (two per edge) and at most $|V| - 1$ calls to `Union`.

With a disjoint-set forest, the $O(E)$ operations cost $O(E\,\alpha(V))$ in total, which is $O(E)$ for all practical purposes. So the total is

$$O(V) + \Theta(E \lg E) + O(E\,\alpha(V)) = O(E \lg E). \tag{11.9}$$

A natural question is why Equation (11.9) is the same as $\Theta(E \lg V)$, and it is worth answering. Since G is simple and connected,

$$|V| - 1 \le |E| \le \binom{|V|}{2} < |V|^2. \tag{11.10}$$

Taking logarithms of the upper bound in Equation (11.10): $\lg|E| < \lg|V|^2 = 2\lg|V|$. And from the lower bound, $\lg|E| \ge \lg(|V| - 1)$, which is $\lg|V| - o(1)$. So

$$\frac{1}{2}\lg|E| \;<\; \lg|V| \;\le\; \lg|E| + 1,$$

that is, $\lg E = \Theta(\lg V)$. Multiplying by $|E|$: $\Theta(E\lg E) = \Theta(E\lg V)$. **The two bounds are the same bound**, which means Kruskal and Prim-with-a-binary-heap have identical asymptotic cost, and the choice between them must be made on other grounds — §11.11.

> **WHERE THE TIME GOES.** Kruskal's cost is *dominated by the sort*. The graph traversal part is essentially linear. So if the edges arrive already sorted — or if the weights are small integers and you can counting-sort them in $O(E + W)$ — Kruskal drops to $O(E\,\alpha(V))$, which is very nearly linear. That is a real trick, and Exercise 11.14 uses it.

11.10.13 The complete program

Listing 11.5 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`.

```
// 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 11.5 (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 11.5 (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;
}
```

Its output, abridged to the decisions (the full run prints the forest before each one):

```
edges in sorted order:
  (gh)1 (ci)2 (fg)2 (ab)4 (cf)4 (gi)6 (cd)7 (hi)7 (ah)8 (bc)8 (de)9 (ef)10 (bh)11 (df)14

  edge  1: (g,h) w= 1  Find(g)=g Find(h)=h  DIFFERENT -> accept, total=1
  edge  2: (c,i) w= 2  Find(c)=c Find(i)=i  DIFFERENT -> accept, total=3
  edge  3: (f,g) w= 2  Find(f)=f Find(g)=g  DIFFERENT -> accept, total=5
  edge  4: (a,b) w= 4  Find(a)=a Find(b)=b  DIFFERENT -> accept, total=9
  edge  5: (c,f) w= 4  Find(c)=c Find(f)=g  DIFFERENT -> accept, total=13
  edge  6: (g,i) w= 6  Find(g)=c Find(i)=c  SAME -> reject
  edge  7: (c,d) w= 7  Find(c)=c Find(d)=d  DIFFERENT -> accept, total=20
  edge  8: (h,i) w= 7  Find(h)=c Find(i)=c  SAME -> reject
  edge  9: (a,h) w= 8  Find(a)=a Find(h)=c  DIFFERENT -> accept, total=28
  edge 10: (b,c) w= 8  Find(b)=c Find(c)=c  SAME -> reject
  edge 11: (d,e) w= 9  Find(d)=c Find(e)=e  DIFFERENT -> accept, total=37
  edge 12: (e,f) w=10  Find(e)=c Find(f)=c  SAME -> reject
  edge 13: (b,h) w=11  Find(b)=c Find(h)=c  SAME -> reject
  edge 14: (d,f) w=14  Find(d)=c Find(f)=c  SAME -> reject

MST edges, in the order Kruskal accepted them:
  (g,h) w=1
  (c,i) w=2
  (f,g) w=2
  (a,b) w=4
  (c,f) w=4
  (c,d) w=7
  (a,h) w=8
  (d,e) w=9
  edge count = 8  (|V|-1 = 8)
  total weight = 37
  components at the end = 1
```

Every line matches **Table 11.11** and every decision matches the fourteen panels. The `components at the end = 1` line is the mechanical check that the answer really is *spanning*.

11.11 Prim versus Kruskal

11.11.1 The two answers, compared

Both algorithms ran on the same nine-vertex graph and both returned a tree of weight 37. But not the same tree:

$$T_{\text{Prim}} = \{(a,b),(b,c),(c,i),(c,f),(f,g),(g,h),(c,d),(d,e)\}$$
$$T_{\text{Kruskal}} = \{(g,h),(c,i),(f,g),(a,b),(c,f),(c,d),(a,h),(d,e)\}$$

Seven of the eight edges are shared. They differ in exactly one place: Prim took (b,c) of weight 8, Kruskal took (a,h), also of weight 8. That single swap is traceable to the tie in **Figure 11.11** and the rejection in **Figure 11.23**, and §11.12 explains why both are legitimate.

11.11.2 Sparse and dense

Recall Equation (11.10): $|V|-1 \le |E| < |V|^2$. A graph is loosely called **sparse** when $|E|$ is close to the bottom of that range and **dense** when it is close to the top. §11.10.12 showed the two bounds are formally identical — $O(E\lg E) = O(E\lg V)$ — so the difference is in constants and in what each algorithm *touches*.

- Kruskal's cost is one sort of $|E|$ items plus a nearly linear scan. Sorting is sequential, cache-friendly, and $|E|$ is small in a sparse graph.

- Prim's cost is $|V|$ heap extractions plus up to $|E|$ key updates, with random access into an adjacency structure. That is pointer-chasing, and it pays off when $|E|$ is large because Prim never sorts the heavy edges it will never need.
- If the graph is **very** dense, the array version of Prim at $O(V^2)$ beats both, because $O(V^2) < O(V^2 \lg V)$ and it has no heap at all.

11.11.3 Measured

Listing 11.6 runs both on the nine-vertex example graph, then on a thousand random connected graphs to check that they always agree, then on one large sparse and one large dense graph with a clock running.

Listing 11.6 Prim and Kruskal measured against each other, including a thousand-graph randomized agreement test. Compile with `g++ -std=c++17 -O2 -o compare compare.cpp`.

```
// Prim and Kruskal side by side. Both are run on the graph of Section 11.9.5 and
// then on a thousand random connected graphs, and the total weights compared.
#include <iostream>
#include <iomanip>
#include <vector>
#include <queue>
#include <algorithm>
#include <random>
#include <climits>
#include <chrono>
using namespace std;

struct Edge { int u, v, w; };

// ---------- Prim-Jarnik, binary heap ----------
long long prim(int n, const vector<Edge>& edges) {
    // Adjacency in one flat array (compressed sparse row): start[u]..start[u+1]-1
    // are u's entries in nbr/wt. One allocation instead of n of them.
    vector<int> start(n + 1, 0), nbr(2 * edges.size()), wt(2 * edges.size());
    for (const Edge& e : edges) { ++start[e.u]; ++start[e.v]; }
    for (int u = 0, acc = 0; u <= n; ++u) { int c = (u < n ? start[u] : 0); start[u] = acc;
acc += c; }
    { vector<int> fill(start.begin(), start.end() - 1);
      for (const Edge& e : edges) {
          nbr[fill[e.u]] = e.v; wt[fill[e.u]++] = e.w;
          nbr[fill[e.v]] = e.u; wt[fill[e.v]++] = e.w;
      } }
    vector<int>  key(n, INT_MAX);
    vector<char> inQ(n, 1);
    key[0] = 0;
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    Q.push({0, 0});
    long long total = 0;
    int taken = 0;
    while (!Q.empty()) {
        auto [k, u] = Q.top(); Q.pop();
        if (!inQ[u] || k != key[u]) continue;
        inQ[u] = 0; total += k; ++taken;
        for (int j = start[u]; j < start[u + 1]; ++j) {
            int v = nbr[j], w = wt[j];
            if (inQ[v] && w < key[v]) { key[v] = w; Q.push({w, v}); }
        }
    }
    return taken == n ? total : -1;                    // -1 means disconnected
}
```

Listing 11.6 (Continued) *Prim and Kruskal measured against each other, including a thousand-graph randomized agreement test. Compile with g++ -std=c++17 -O2 -o compare compare.cpp.*

```
// ---------- Kruskal, disjoint-set forest ----------
struct DSU {
    vector<int> p, r;
    explicit DSU(int n) : p(n), r(n, 0) { for (int i = 0; i < n; ++i) p[i] = i; }
    int find(int x) { return p[x] == x ? x : p[x] = find(p[x]); }
    bool unite(int x, int y) {
        x = find(x); y = find(y);
        if (x == y) return false;
        if (r[x] < r[y]) swap(x, y);
        p[y] = x;
        if (r[x] == r[y]) ++r[x];
        return true;
    }
};

long long kruskal(int n, vector<Edge> edges) {
    sort(edges.begin(), edges.end(), [](const Edge& a, const Edge& b) { return a.w < b.w;
});
    DSU D(n);
    long long total = 0;
    int taken = 0;
    for (const Edge& e : edges)
        if (D.unite(e.u, e.v)) { total += e.w; ++taken; }
    return taken == n - 1 ? total : -1;
}

int main() {
    // The example graph: a=0 b=1 c=2 d=3 e=4 f=5 g=6 h=7 i=8
    vector<Edge> example = {
        {0,1,4}, {0,7,8}, {1,2,8}, {1,7,11}, {2,3,7}, {2,5,4}, {2,8,2},
        {3,4,9}, {3,5,14}, {4,5,10}, {5,6,2}, {6,7,1}, {6,8,6}, {7,8,7}
    };
    cout << "the example graph: V=9, E=" << example.size() << "\n";
    cout << "  Prim    total = " << prim(9, example)    << "\n";
    cout << "  Kruskal total = " << kruskal(9, example) << "\n";
    cout << "  agree: " << (prim(9, example) == kruskal(9, example) ? "yes" : "NO") <<
"\n\n";
```

Listing 11.6 (Continued) *Prim and Kruskal measured against each other, including a thousand-graph randomized agreement test. Compile with g++ -std=c++17 -O2 -o compare compare.cpp.*

```
    mt19937 rng(20251111);
    int disagreements = 0, trials = 1000;
    for (int t = 0; t < trials; ++t) {
        int n = 5 + (int)(rng() % 26);                  // 5..30 vertices
        vector<Edge> es;
        for (int v = 1; v < n; ++v)                     // a random tree keeps it connected
            es.push_back({(int)(rng() % v), v, 1 + (int)(rng() % 20)});
        int extra = (int)(rng() % (2 * n + 1));
        for (int k = 0; k < extra; ++k) {
            int a = (int)(rng() % n), b = (int)(rng() % n);
            if (a != b) es.push_back({a, b, 1 + (int)(rng() % 20)});
        }
        if (prim(n, es) != kruskal(n, es)) ++disagreements;
    }
    cout << trials << " random connected graphs (5..30 vertices, weights 1..20)\n";
    cout << "  disagreements between Prim and Kruskal: " << disagreements << "\n\n";

    // Timing on one larger dense graph and one larger sparse graph.
    auto build = [&](int n, long long m) {
        vector<Edge> es;
        for (int v = 1; v < n; ++v) es.push_back({(int)(rng() % v), v, 1 + (int)(rng() %
1000)});
        while ((long long)es.size() < m) {
            int a = (int)(rng() % n), b = (int)(rng() % n);
            if (a != b) es.push_back({a, b, 1 + (int)(rng() % 1000)});
        }
        return es;
    };
    cout << fixed << setprecision(1);
    cout << left << setw(10) << "shape" << setw(10) << "V" << setw(12) << "E"
         << setw(12) << "Prim ms" << setw(12) << "Kruskal ms" << "weights equal\n";
    struct Shape { const char* tag; int n; long long m; };
    for (Shape s : { Shape{"sparse", 200000, 400000}, Shape{"dense", 2000, 1000000} }) {
        vector<Edge> es = build(s.n, s.m);
        auto t0 = chrono::steady_clock::now();
        long long wp = prim(s.n, es);
        auto t1 = chrono::steady_clock::now();
        long long wk = kruskal(s.n, es);
        auto t2 = chrono::steady_clock::now();
        cout << left << setw(10) << s.tag << setw(10) << s.n << setw(12) << es.size()
             << setw(12) << chrono::duration<double, milli>(t1 - t0).count()
             << setw(12) << chrono::duration<double, milli>(t2 - t1).count()
             << (wp == wk ? "yes" : "NO") << "\n";
    }
    return 0;
}
```

Its output:

```
the example graph: V=9, E=14
  Prim    total = 37
  Kruskal total = 37
  agree: yes

1000 random connected graphs (5..30 vertices, weights 1..20)
  disagreements between Prim and Kruskal: 0

shape     V         E           Prim ms     Kruskal ms  weights equal
sparse    200000    400000      104.9       26.3        yes
dense     2000      1000000     39.9        58.7        yes
```

Three things are worth taking from that. First, the cross-check: both algorithms returned 37 on the example graph, and agreed on all one thousand random graphs — which is what Theorem

11.5 guarantees and is a good regression test for any implementation you write. Second, the shapes behave as predicted: on the sparse graph Kruskal is about four times faster, and on the dense graph Prim is about a third faster. Third, the four millisecond figures are one run, on one machine, with one compiler. Run the program again and they will move by a few percent; run it on your own machine and they will move by more. The *ratio* is the transferable fact — Kruskal roughly four times faster on the sparse graph, Prim roughly a third faster on the dense one — and even the ratio will shift if you change the weight distribution or the adjacency layout. Never quote a timing without saying what produced it, and never let a test assert one.

11.11.4 The decision table

Table 11.12 collects the advice, with the reason in every row, so that the choice can be defended rather than merely made.

Table 11.12 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
$\lvert E \rvert = \Theta(V^2)$ *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 $\lvert V \rvert - 1$ edges
you need the tree rooted at a particular vertex, with parents	**Prim**	π is the rooted tree, for free
$\lvert E \rvert = \Theta(V \lg V)$ or denser and V is in the millions	**Prim**, Fibonacci heap	$O(V \lg V + E)$ is the best known bound

> **THE HONEST SUMMARY.** Kleinberg and Tardos [12] make the point that both algorithms are instances of one idea, and that what you choose between is the implementation, not the idea. For most problems either will do, and the one you should use is the one you can write correctly from memory under time pressure. Kruskal is usually that one: sort, then a ten-line loop over a disjoint-set structure you already have. Reach for Prim when the graph is dense, when it is handed to you as a matrix, or when you want the tree rooted.

11.12 Equal Weights, and When the MST Is Unique

11.12.1 Ties break the algorithms apart, not the answer

We have already seen the mechanism twice. In **Figure 11.11** Prim faced c: 8 and h: 8 and had to pick one; in **Figure 11.23** Kruskal faced (a, h)8 and (b, c)8 and accepted whichever came first in the sorted order. Neither choice is wrong, and the reason is the word *some* in Theorem 11.5: when two crossing edges tie for lightest, **each of them lies in some MST**, and those may be different MSTs of equal weight.

So with ties present, the *tree* an algorithm returns depends on implementation details: the comparator you passed to `sort`, whether your heap is a max-heap of negated keys or a min-heap, the order you listed the edges in. The *weight* does not. That is the guarantee, and it is the one to test against.

A TRAP FOR TEST SUITES. Never assert that an MST implementation returns a particular edge set. Assert that (a) the result has $|V| - 1$ edges, (b) it is connected and acyclic, and (c) its weight equals a reference weight. **Listing 11.6** tests exactly (c), across a thousand graphs, and that is why it finds zero disagreements rather than a thousand.

11.12.2 Distinct weights imply a unique MST

THEOREM 11.8. If all edge weights of a connected graph G are **distinct**, then G has exactly one minimum spanning tree.

Proof. Suppose T_1 and T_2 are both MSTs and $T_1 \neq T_2$. Then some edge lies in one and not the other; among all such edges, let e be the one of **smallest** weight, and say without loss of generality $e \in T_1 \setminus T_2$.

Add e to T_2. By §11.4.2 this creates exactly one cycle C, and C contains e. Now C cannot lie entirely inside T_1, because T_1 is acyclic. So C contains some edge $f \notin T_1$, with $f \neq e$ and $f \in T_2$. So $f \in T_2 \setminus T_1$, which makes f one of the edges over which e was chosen minimal. Since the weights are distinct and e was the minimum such edge,

$$w(e) < w(f).$$

Now form $T_2{}' = (T_2 \cup \{e\}) \setminus \{f\}$. Removing an edge of the unique cycle C leaves a spanning tree (the cycle is broken, the vertex count is unchanged, the edge count is back to $|V| - 1$, and connectivity survives because the rest of C provides the missing route). And

$$w(T_2{}') = w(T_2) + w(e) - w(f) < w(T_2).$$

That contradicts the minimality of T_2. Hence no such pair $T_1 \neq T_2$ exists. ■

Note the converse is **false**. A graph can have repeated weights and still have a unique MST — if the tied edges never compete for the same cut. The complete graph on three vertices with weights 1, 1, 5 has a unique MST (the two 1s), despite the tie.

11.12.3 Testing uniqueness

Here is an exact test, and it is only a few lines given Kruskal.

CRITERION. Let T be an MST of G. Then T is the *only* MST of G if and only if, for every edge $e \in T$, the minimum spanning tree of $G - e$ (the graph with e deleted) is strictly heavier than T.

Why. ($\Leftarrow$) Suppose some other MST $T' \neq T$ exists. Pick $e \in T \setminus T'$, which is non-empty since the two trees differ and both have $|V| - 1$ edges. Then T' is a spanning tree of $G - e$ of weight $w(T)$, so $\mathrm{MST}(G - e) \leq w(T)$; and $\mathrm{MST}(G - e) \geq w(T)$ always, since $G - e$ has fewer trees to choose from than G. So $\mathrm{MST}(G - e) = w(T)$ — not strictly heavier. ($\Rightarrow$) Conversely, if $\mathrm{MST}(G - e) = w(T)$ for some $e \in T$, then that tree is an MST of G not containing e, hence different from T. ■

Running that test on the nine-vertex example graph — Kruskal called eight times, once per tree edge — gives:

Table 11.13 The uniqueness test of §11.12.3 applied to the nine-vertex example graph, computed by calling Kruskal from **Listing 11.5** 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

So the example graph has **exactly two** minimum spanning trees, of weight 37 each, and the two algorithms of this chapter happen to find one each. (The test shows that the MST is not unique; that there are no more than two follows because only the edge (a,h) can be dropped at no cost, and the only replacement of equal weight is (b,c) — a brute-force enumeration of all spanning trees confirms it.) That is not a coincidence engineered for the example; it is what happens when a graph has tied weights competing across the same cut, and it is the cleanest possible illustration of the word *some* in Theorem 11.5. (The same program, run with the sort reversed, returns the **maximum** spanning tree, of weight 71 — Kruskal works just as well upside down, because the cut property has an exact dual for maxima. Exercise 11.12.)

11.13 Profitina in Practice

Profitina is an **AI-visibility (GEO) platform**: it scans a website, scores how ready its content is to be *cited* by an AI answer engine, explains each signal, and helps fix what it finds. It runs in production at `app.profitina.com`, as a Python application over PostgreSQL, with local model inference through Ollama and Qwen2.5. Two of this chapter's ideas are load-bearing in it.

11.13.1 The scan pipeline is a DAG, and the scheduler topologically sorts it

A single site scan is not one measurement; it is a few dozen, and they depend on one another. Fetching a page must precede parsing it. Parsing must precede the content-extractability signals, the structured-data signals and the internal-link signals. Canonicalizing URLs must precede deduplicating near-identical pages, which must precede any per-page aggregate. Resolving `robots.txt` must precede deciding which pages are eligible at all. And the final readiness score depends on every signal beneath it.

That is a dependency DAG, and it is handled exactly as §11.3 describes. The stage graph is checked for cycles at start-up — a cycle would mean a signal that depends on itself, which is a configuration bug and must fail loudly rather than deadlock — and then topologically sorted to give a legal execution order. Kahn's algorithm is the right one here, for the reason in **Table 11.5**: the ready set is a natural work queue, so stages whose dependencies are all satisfied can be handed to workers immediately and in parallel, and new stages become ready as results arrive.

The DFS variant would give a single sequential order and throw away the parallelism the DAG is telling you about.

11.13.2 Minimum spanning trees in near-duplicate detection

A large site has many pages that are almost the same page: a product template filled with different products, a paginated archive, a print view. Scoring each of them separately wastes work and distorts the aggregate. So near-identical pages must be grouped.

Treat each page as a vertex and each pair as an edge weighted by *distance* — a small number when two pages are nearly identical, large when they differ. Build a minimum spanning tree of that graph and then delete every edge heavier than a threshold. What remains is a set of components, and those components are the duplicate groups. This is **single-linkage clustering**, and the connection is exact: the clusters produced by cutting an MST at a threshold τ are precisely the single-linkage clusters at τ. Kruskal makes it even more direct — it *is* single-linkage clustering, run from the cheapest edge up, and stopping it early at τ gives the clustering without ever finishing the tree.

11.13.3 The worked case study: Profitina Laundry

The runnable example is always **Profitina Laundry**, the teaching case study, never the real product. Profitina Laundry has ten branches around Surabaya. Two questions, one from each half of this chapter.

Question 1. The company wants a private shuttle network so that a garment dropped at any branch can reach any other branch — for consolidating loads, moving overflow, or delivering a customer's order to a branch nearer their home. The operations team has driven eighteen candidate routes between pairs of branches and recorded the distance of each. Laying on a route costs money per kilometer driven, every day, forever. **Which routes should be operated?** By §11.5.2 this is exactly the MST problem: the network must span all ten branches, any cycle in it contains a route that could be dropped without disconnecting anything, and we want the least total distance.

Question 2. A laundry job passes through nine stages with real precedence constraints, and each stage takes a known number of minutes. **In what order may the stages be run, and how long does a job take at best?** The first part is a topological sort; the second is the longest chain through the DAG, which — because the vertices come out in topological order — can be computed in the same pass, with one relaxation per edge. (That longest chain is the *critical path*: the sequence of stages that determines the job's minimum turnaround. Chapter 13 treats this properly as a shortest-path problem on a DAG; here it is a two-line addition to Kahn's algorithm.)

Listing 11.7 The Profitina Laundry case study: a minimum spanning tree over ten branches, and a topological sort with a critical path over nine job stages. Compile with `g++ -std=c++17 -O2 -o laundry laundry.cpp`.

```
// Profitina Laundry: two problems from one chapter.
//   Part 1 - the cheapest set of shuttle links that keeps ten branches connected
//            (a minimum spanning tree), solved by Kruskal and checked by Prim.
//   Part 2 - a legal order in which to run the nine stages of a laundry job
//            (a topological sort), and the length of the longest chain.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
using namespace std;

// ============================ Part 1: the shuttle network ==================
struct Link { int a, b, km100; };                 // distance in hundreds of meters

struct DSU {
    vector<int> parent, rank_;
    explicit DSU(int n) : parent(n), rank_(n, 0) {
        for (int i = 0; i < n; ++i) parent[i] = i;
    }
    int find(int x) { return parent[x] == x ? x : parent[x] = find(parent[x]); }
    bool unite(int x, int y) {
        x = find(x); y = find(y);
        if (x == y) return false;
        if (rank_[x] < rank_[y]) swap(x, y);
        parent[y] = x;
        if (rank_[x] == rank_[y]) ++rank_[x];
        return true;
    }
};

int primTotal(int n, const vector<Link>& links) {
    vector<vector<pair<int,int>>> adj(n);
    for (const Link& l : links) {
        adj[l.a].push_back({l.b, l.km100});
        adj[l.b].push_back({l.a, l.km100});
    }
    vector<int> key(n, INT_MAX); vector<char> inQ(n, 1);
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    key[0] = 0; Q.push({0, 0});
    int total = 0, taken = 0;
    while (!Q.empty()) {
        auto [k, u] = Q.top(); Q.pop();
        if (!inQ[u] || k != key[u]) continue;
```

Listing 11.7 (Continued) *The Profitina Laundry case study: a minimum spanning tree over ten branches, and a topological sort with a critical path over nine job stages. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
        inQ[u] = 0; total += k; ++taken;
        for (auto [v, w] : adj[u]) if (inQ[v] && w < key[v]) { key[v] = w; Q.push({w, v});
}
    }
    return taken == n ? total : -1;
}

void part1() {
    vector<string> branch = { "Keputih", "Gebang", "Klampis", "Manyar", "Ngagel",
                              "Darmo", "Wonokromo", "Rungkut", "Gubeng", "Kertajaya" };
    // Road distances the operations team actually drove, in hundreds of meters.
    vector<Link> links = {
        {0,1,12}, {0,7,18}, {1,2,9},  {1,9,7},  {2,3,14}, {2,9,6},
        {3,4,11}, {3,8,8},  {4,5,10}, {4,6,13}, {5,6,7},  {6,7,22},
        {7,9,25}, {8,9,5},  {8,4,9},  {0,9,20}, {5,8,16}, {1,7,15}
    };
    const int n = (int)branch.size();

    int allLinks = 0;
    for (const Link& l : links) allLinks += l.km100;

    sort(links.begin(), links.end(),
         [](const Link& x, const Link& y) { return x.km100 < y.km100; });

    DSU D(n);
    vector<Link> chosen;
    int total = 0;
    cout << "Part 1 - cheapest shuttle network over " << n << " branches, "
         << links.size() << " candidate links\n";
    cout << "  link                                  km    decision\n";
    for (const Link& l : links) {
        bool take = (D.find(l.a) != D.find(l.b));
        if (take) { D.unite(l.a, l.b); chosen.push_back(l); total += l.km100; }
        cout << "  " << left << setw(12) << branch[l.a] << "- " << setw(12) << branch[l.b]
             << right << setw(8) << fixed << setprecision(1) << l.km100 / 10.0
             << "    " << (take ? "lay it" : "skip (already reachable)") << "\n";
    }
    cout << "  links laid        : " << chosen.size() << "  (needed: " << n - 1 << ")\n";
    cout << "  network length    : " << total / 10.0 << " km\n";
    cout << "  every link laid   : " << allLinks / 10.0 << " km\n";
    cout << "  saved             : " << (allLinks - total) / 10.0 << " km ("
         << 100.0 * (allLinks - total) / allLinks << "% of the candidate length)\n";
    int byPrim = primTotal(n, links);
    cout << "  Prim's answer     : " << byPrim / 10.0 << " km -- "
         << (byPrim == total ? "identical" : "DIFFERENT") << "\n\n";
}
```

Listing 11.7 (Continued) *The Profitina Laundry case study: a minimum spanning tree over ten branches, and a topological sort with a critical path over nine job stages. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
// ========================== Part 2: the job workflow ====================
void part2() {
    vector<string> stage = { "intake", "weigh", "price", "sort", "wash",
                             "dry", "press", "fold", "handover" };
    vector<int>    minutes = { 4, 3, 2, 6, 35, 40, 12, 8, 3 };
    enum { INTAKE, WEIGH, PRICE, SORT, WASH, DRY, PRESS, FOLD, HANDOVER };
    const int n = (int)stage.size();
    vector<vector<int>> next_(n);
    auto edge = [&](int u, int v) { next_[u].push_back(v); };
    edge(INTAKE, WEIGH);  edge(INTAKE, SORT);
    edge(WEIGH, PRICE);   edge(PRICE, HANDOVER);
    edge(SORT, WASH);     edge(WASH, DRY);
    edge(DRY, PRESS);     edge(DRY, FOLD);
    edge(PRESS, FOLD);    edge(FOLD, HANDOVER);

    vector<int> indeg(n, 0);
    for (int u = 0; u < n; ++u) for (int v : next_[u]) ++indeg[v];
    queue<int> ready;
    for (int u = 0; u < n; ++u) if (indeg[u] == 0) ready.push(u);

    vector<int> order;
    while (!ready.empty()) {
        int u = ready.front(); ready.pop();
        order.push_back(u);
        for (int v : next_[u]) if (--indeg[v] == 0) ready.push(v);
    }

    cout << "Part 2 - a legal order for the " << n << " stages\n  ";
    for (int u : order) cout << stage[u] << (u == order.back() ? "\n" : " -> ");

    // Longest chain: relax each stage in topological order. This is the
    // earliest moment every stage can finish if nothing ever waits for a machine.
    vector<int>    finish(n, 0);
    vector<int>    from(n, -1);
    for (int u : order) {
        if (finish[u] == 0) finish[u] = minutes[u];
        for (int v : next_[u])
            if (finish[u] + minutes[v] > finish[v]) {
                finish[v] = finish[u] + minutes[v];
                from[v]   = u;
            }
    }
    int last = (int)(max_element(finish.begin(), finish.end()) - finish.begin());
    cout << "  earliest finish per stage:\n";
    for (int u : order)
        cout << "    " << left << setw(10) << stage[u] << right << setw(4)
```

Listing 11.7 (Continued) *The Profitina Laundry case study: a minimum spanning tree over ten branches, and a topological sort with a critical path over nine job stages. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
             << finish[u] << " min\n";
    vector<string> chain;
    for (int v = last; v != -1; v = from[v]) chain.push_back(stage[v]);
    reverse(chain.begin(), chain.end());
    cout << "  longest chain (" << finish[last] << " min): ";
    for (size_t i = 0; i < chain.size(); ++i) cout << (i ? " -> " : "") << chain[i];
    cout << "\n";
}

int main() { part1(); part2(); return 0; }
```

Its output:

```
Part 1 - cheapest shuttle network over 10 branches, 18 candidate links
  link                                  km    decision
  Gubeng      - Kertajaya        0.5    lay it
  Klampis     - Kertajaya        0.6    lay it
  Gebang      - Kertajaya        0.7    lay it
  Darmo       - Wonokromo        0.7    lay it
  Manyar      - Gubeng           0.8    lay it
  Gebang      - Klampis          0.9    skip (already reachable)
  Gubeng      - Ngagel           0.9    lay it
  Ngagel      - Darmo            1.0    lay it
  Manyar      - Ngagel           1.1    skip (already reachable)
  Keputih     - Gebang           1.2    lay it
  Ngagel      - Wonokromo        1.3    skip (already reachable)
  Klampis     - Manyar           1.4    skip (already reachable)
  Gebang      - Rungkut          1.5    lay it
  Darmo       - Gubeng           1.6    skip (already reachable)
  Keputih     - Rungkut          1.8    skip (already reachable)
  Keputih     - Kertajaya        2.0    skip (already reachable)
  Wonokromo   - Rungkut          2.2    skip (already reachable)
  Rungkut     - Kertajaya        2.5    skip (already reachable)
  links laid        : 9  (needed: 9)
  network length    : 7.9 km
  every link laid   : 22.7 km
  saved             : 14.8 km (65.2% of the candidate length)
  Prim's answer     : 7.9 km -- identical

Part 2 - a legal order for the 9 stages
  intake -> weigh -> sort -> price -> wash -> dry -> press -> fold -> handover
  earliest finish per stage:
    intake       4 min
    weigh        7 min
    sort        10 min
    price        9 min
    wash        45 min
    dry         85 min
    press       97 min
    fold       105 min
    handover   108 min
  longest chain (108 min): intake -> sort -> wash -> dry -> press -> fold -> handover
```

Read Part 1 as a manager would. Nine links are operated out of eighteen surveyed, and nine is exactly $|V| - 1$ for ten branches. The network is 7.9 km against 22.7 km for operating everything — a 65 % reduction in daily shuttle distance, with every branch still reachable from every other. Half the skipped links are cheap ones (0.9 km, 1.1 km) that happened to be redundant, which is the whole point: cheapness is not the criterion, *usefulness* is, and the disjoint-set structure is what tells the two apart. Prim, run on the same data, returns 7.9 km — the cross-check.

Read Part 2 as an operations planner would. The order produced is one of many legal ones, and the critical path is `intake → sort → wash → dry → press → fold → handover`, 108 minutes. Notice which stages are *not* on it: `weigh` and `price` finish at minute 7 and 9 and then wait 99 minutes for the linen. So buying a second scale would shorten nothing, and shaving five minutes off the dryer cycle would shorten everything by five minutes. That is the practical value of the critical path, and it fell out of a topological sort.

11.14 Common Mistakes, and How to Avoid Them

1. Confusing a minimum spanning tree with a shortest-path tree. The MST minimizes the *total* weight of the network. Dijkstra's tree minimizes the distance *from one source to each vertex*.

They are different objectives and usually different trees. In Prim's MST of the example graph (**Figure 11.17**), the tree route from a to e is $a - b - c - d - e$, costing $4 + 8 + 7 + 9 = 28$, while $a - h - g - f - e$ costs $8 + 1 + 2 + 10 = 21$. *Avoid it by* asking what you are minimizing before you choose the algorithm: a sum over the whole tree, or a distance to each vertex.

2. Writing `key[u] + w(u,v) < key[v]` in Prim. That is Dijkstra's relaxation; Prim's is `w(u,v) < key[v]`. There is no accumulated path length in an MST. *Avoid it by* remembering that $key[v]$ is the weight of one edge, never of a path.

3. Forgetting the `v in Q` test in Prim. Without it, a vertex already in the tree can have its key lowered, its π overwritten, and a duplicate edge added. *Avoid it by* treating Q-membership as the single source of truth about who is inside the tree.

4. Not sorting in Kruskal, or sorting descending. §11.10.11 showed the correctness proof rests entirely on "any cheaper crossing edge would already have been examined". Descending order gives you the *maximum* spanning tree, which is a fine thing to want and a disaster to produce by accident. *Avoid it by* printing the first five weights of your sorted list during development.

5. Using a plain array or a `set<int>` for Kruskal's component test. Scanning a component to see whether v is in it is $O(V)$ per edge, turning $O(E\lg E)$ into $O(VE)$. *Avoid it by* using a disjoint-set forest. It is fifteen lines.

6. Omitting union by rank *or* path compression and expecting α. Path compression alone gives $O(\lg n)$ amortized; union by rank alone gives $O(\lg n)$ worst case. Only both together give $O(\alpha(n))$. *Avoid it by* writing them as one class, as in **Listing 11.4**, so you can never have one without the other.

7. Asserting a particular MST in a test. With tied weights there are several correct answers (**Table 11.13** exhibits two). *Avoid it by* asserting the *weight*, the edge count $|V| - 1$, and connectivity.

8. Pushing onto the back of the list in DFS topological sort. That gives increasing finishing time, which is the reverse of a topological order. *Avoid it by* checking one edge of the output by hand before believing it.

9. Recording a vertex at discovery time rather than at finishing time. The discovery order is not a topological order; in **Table 11.3** it starts `shirt, tie, jacket`, and `jacket` has two unlisted predecessors. *Avoid it by* putting the `push_front` after the loop over $Adj[u]$, never before.

10. Assuming a topological sort is unique. It almost never is; the two algorithms of §11.3 disagree completely on the same graph. *Avoid it by* verifying the property, not the string.

11. Applying the DAG theorem to an undirected graph without ignoring the arriving edge. The trap of §11.2.5. *Avoid it by* passing the parent down the recursion.

12. Running Prim on a disconnected graph and believing the answer. Prim stops when its queue is exhausted of reachable vertices and silently returns a tree of the component containing the root. *Avoid it by* counting the extracted vertices and comparing with $|V|$ — **Listing 11.6** returns -1 for exactly this reason.

13. Overflowing on weights. Summing $|V| - 1$ edge weights of up to 10^9 each overflows a 32-bit `int` as soon as $|V| = 4$: three such edges already sum to $3 \times 10^9 > 2^{31} - 1 \approx 2.1 \times 10^9$. *Avoid it by* accumulating totals in `long long`, as **Listing 11.6** does.

11.15 Chapter Summary

- A **directed acyclic graph** is a directed graph with no cycle. It models precedence: build dependencies, course prerequisites, task schedules, instruction dependencies.
- **Theorem 11.1 (the DAG theorem):** G is acyclic if and only if a depth-first search of G yields no back edge. The proof has two halves — a back edge closes a cycle with the tree path; and a cycle cannot hide, because the first-discovered vertex of the cycle sees a white path all the way round. Cycle detection therefore costs $\Theta(V + E)$.
- A **topological sort** is a linear order of the vertices in which every edge points forward. It exists exactly for DAGs, and it is almost never unique.
- Two algorithms, both $\Theta(V + E)$: list the vertices in **decreasing finishing time** of a DFS (correct because $f[v] < f[u]$ for every edge (u, v), proved by a three-case argument); or repeatedly remove a vertex of **in-degree 0** (Kahn), which detects cycles by the shortfall $|L| < |V|$.
- A **spanning tree** contains every vertex, is connected, and has no cycle; by **Theorem 11.3** it has exactly $|V| - 1$ edges, and adding any further edge creates exactly one cycle.
- A **minimum spanning tree** is a spanning tree of least total weight. Brute force is hopeless: a complete graph on n vertices has n^{n-2} spanning trees.
- **Theorem 11.4 (optimal substructure):** removing an edge from an MST leaves two trees, each an MST of the subgraph induced by its own vertices. Proved by cut-and-paste.
- **Theorem 11.5 (the cut property):** for any cut $(S, V - S)$, a minimum-weight crossing edge belongs to *some* MST. Proved by exchange: the path the MST provides must cross the cut somewhere, that crossing cannot be cheaper than ours, so swap. **Corollary 11.6** extends it to any cut respecting a partial answer A — which is what turns the theorem into an algorithm.
- `Generic-MST` repeatedly adds a **safe edge**; the loop invariant "A extends to an MST" makes it correct. The only freedom is which cut to use, and that freedom is the difference between the two algorithms.
- **Prim–Jarník** grows one tree from a root, keeping $key[v]$ = the cheapest edge from v into the tree and $\pi[v]$ = its other end, in a priority queue. $O(V^2)$ with an array, $O(E\lg V)$ with a binary heap, $O(V\lg V + E)$ with a Fibonacci heap.
- **Kruskal** sorts all edges and accepts each one whose endpoints lie in different components, using a **disjoint-set forest** with **union by rank** (Lemma 11.7: rank k means at least 2^k elements, so height $\leq \lg n$) and **path compression**. Total $O(E\lg E) = O(E\lg V)$, dominated by the sort; the set operations cost $O(E\,\alpha(V))$, which is effectively linear.
- The two produce the same *weight* always, and the same *tree* only when the MST is unique. **Theorem 11.8:** distinct weights imply a unique MST. The converse fails.
- Choose Kruskal for sparse graphs, streams, and already-sorted or small-integer weights; Prim for dense graphs, adjacency matrices, and when you want the tree rooted.

11.15.1 Looking ahead

Chapter 13 asks the other weighted-graph question — the cheapest route between two places — and you will meet Dijkstra's algorithm, which looks so much like Prim that the difference is one `+` sign, and is a different algorithm solving a different problem. It also solves the shortest-path problem on a DAG in $\Theta(V + E)$ by relaxing edges in the topological order computed here, which is the honest version of the critical path we computed in **Listing 11.7**. **Chapter 14** owes us the proof that path compression with union by rank costs $O(\alpha(n))$ amortized, and the potential method it develops is also what justifies the Fibonacci-heap row of **Table 11.8**. **Chapter 15** takes the shortest-path question to all pairs at once.

11.16 Exercises

11.1 Run the DFS topological sort by hand on the DAG with vertices {1,2,3,4,5} and edges 1 → 2, 1 → 3, 2 → 4, 3 → 4, 4 → 5, visiting vertices in the outer loop in index order and each adjacency list in ascending order. Give d, f and the output order.

11.2 How many distinct topological orders does the DAG of Exercise 11.1 have? List them all.

11.3 Explain why the test "there is a back edge" reports a cycle in *every* undirected graph with at least one edge, and give the one-line repair. Write the repaired recursive function.

11.4 Prove that every finite DAG has at least one vertex of in-degree 0, and at least one of out-degree 0. Where in §11.3.11 is the first of these used?

11.5 Run Prim on the nine-vertex example graph with root h instead of a. Give the order of extraction and the total weight. Is the tree the same as **Figure 11.17**?

11.6 In **Table 11.11**, edge 6 is (g, i) of weight 6 and it is rejected. Name the cycle it would have created, and check that every other edge of that cycle is lighter than 6.

11.7 Prove: a connected graph on n vertices with exactly $n - 1$ edges is a tree. (This is the converse used in the proof of Theorem 11.4.)

11.8 Take the triangle $a - b$ of weight 1, $b - c$ of weight 1, $a - c$ of weight 1.5. Compute (i) the MST, and (ii) the shortest-path tree rooted at a. Show they differ, and identify the one line of Prim that would have to change to produce the second.

11.9 Using the criterion of §11.12.3, show by hand that the graph with vertices $\{a, b, c\}$ and edges $a - b = 1$, $b - c = 1$, $a - c = 1$ has three distinct MSTs, and that no graph with all-distinct weights can have two.

11.10 The **second-best MST** of G is a spanning tree of minimum weight among all spanning trees other than the MST. Give an $O(VE)$ algorithm, and argue that the second-best tree differs from the best by exactly one edge swap.

11.11 The **bottleneck** of a spanning tree is the weight of its heaviest edge. Prove that a minimum spanning tree also minimizes the bottleneck over all spanning trees. Is the converse true?

11.12 State and prove the dual of Theorem 11.5 for **maximum** spanning trees, and say what one line of **Listing 11.5** must change. Verify your answer by finding the maximum spanning tree of the example graph by hand.

11.13 Replace the lazy `DecreaseKey` of **Listing 11.3** with an **indexed binary heap**: keep `pos[v]`, the index of v in the heap array, updated by every sift. Give the resulting running time, and say how much space it saves.

11.14 Suppose every edge weight is an integer in $[1, W]$ with $W = O(E)$. Show how to run Kruskal in $O(E\,\alpha(V))$ time, and say what changes in the correctness proof (if anything).

11.15 Let G be connected with weight function w, and let $w'(u, v) = w(u, v) + c$ for a constant $c > 0$. Prove that every MST under w is an MST under w'. Then show by a three-vertex example that the corresponding statement for *shortest paths* is false.

11.16 True or false, with proof or counterexample: (a) the lightest edge of G is in every MST; (b) the heaviest edge of G is in no MST; (c) if an edge is the unique lightest edge across some cut, it is in every MST.

11.17 Modify Kruskal to compute a **minimum spanning forest** of a possibly disconnected graph. What must change in the code, and what must change in the proof of §11.10.11?

11.18 Explain, in your own words and without proving the α bound, why union by rank alone gives $O(\lg n)$ per operation, why path compression alone also gives $O(\lg n)$ amortized, and why the two together do far better than either. Which chapter proves the combined bound?

11.17 Solutions to the Exercises

11.1 DFS from 1: $d[1] = 1$; then 2 ($d = 2$); then 4 ($d = 3$); then 5 ($d = 4, f = 5$); $f[4] = 6$; $f[2] = 7$; back at 1, go to 3 ($d = 8$); 4 is black, so $f[3] = 9$; $f[1] = 10$. Times: 1: 1/10, 2: 2/7, 3: 8/9, 4: 3/6, 5: 4/5. Decreasing f: 1,3,2,4,5. Check: 1 → 2 (pos 1 < 3) ✓, 1 → 3 (1 < 2) ✓, 2 → 4 (3 < 4) ✓, 3 → 4 (2 < 4) ✓, 4 → 5 ✓.

11.2 Two. Vertex 1 must come first and 5 last; 4 must follow both 2 and 3; so the only freedom is the order of 2 and 3. The orders are 1,2,3,4,5 and 1,3,2,4,5.

11.3 In an undirected graph every edge $\{u, v\}$ appears in both $Adj[u]$ and $Adj[v]$. Coming to v from u, the search immediately examines (v, u) and finds u gray — but that is the same edge seen from the other end, not a cycle. The repair is to pass the parent down the recursion and skip it *once*, exactly as the function `hasCycle` in **Listing 11.8** does.

Setting `parent = -1` after the first skip is what makes a *parallel* edge between u and v correctly report a cycle, which it is. **Listing 11.8** puts the broken test and the repaired one side by side on seven small graphs.

Listing 11.8 The broken directed test and the repaired undirected one, measured against seven graphs whose answers are known. Compile with `g++ -std=c++17 -O2 -o undirected undirected.cpp`.

```
// Exercise 11.3: cycle detection in an UNDIRECTED graph.
// The directed test "a gray neighbor means a cycle" fires on every edge,
// because each edge appears in both endpoints' adjacency lists. The repair is
// to ignore the edge we arrived on -- once, so that a parallel edge still counts.
#include <iostream>
#include <string>
#include <vector>
using namespace std;

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); }
};

// The broken test: report a cycle whenever a neighbor has already been seen.
bool brokenHasCycle(const Graph& G, vector<char>& seen, int u) {
    seen[u] = 1;
    for (int v : G.adj[u]) {
        if (seen[v]) return true;                    // fires on the arriving edge too
        if (brokenHasCycle(G, seen, v)) return true;
    }
    return false;
}

// The repair: skip the arriving edge exactly once. Setting parent to -1 after
// the first skip means a second copy of the same edge (a parallel edge) is
// correctly reported as a cycle.
bool hasCycle(const Graph& G, vector<char>& seen, int u, int parent) {
    seen[u] = 1;
    for (int v : G.adj[u]) {
        if (v == parent) { parent = -1; continue; }   // skip the arriving edge once only
        if (seen[v]) return true;                    // a genuine back edge
        if (hasCycle(G, seen, v, u)) return true;
    }
    return false;
}
```

Listing 11.8 (Continued) *The broken directed test and the repaired undirected one, measured against seven graphs whose answers are known. Compile with g++ -std=c++17 -O2 -o undirected undirected.cpp.*

```
bool anyCycle(const Graph& G, bool repaired) {
    vector<char> seen(G.n, 0);
    for (int u = 0; u < G.n; ++u)
        if (!seen[u]) {
            if (repaired ? hasCycle(G, seen, u, -1) : brokenHasCycle(G, seen, u))
                return true;
        }
    return false;
}

int main() {
    struct Case { string name; int n; vector<pair<int,int>> es; bool truth; };
    vector<Case> cases = {
        { "single edge 0-1",            2, {{0,1}},                       false },
        { "path 0-1-2-3",               4, {{0,1},{1,2},{2,3}},           false },
        { "triangle 0-1-2-0",           3, {{0,1},{1,2},{2,0}},           true  },
        { "star, center 0",             4, {{0,1},{0,2},{0,3}},           false },
        { "two parallel edges 0=1",     2, {{0,1},{0,1}},                 true  },
        { "forest: 0-1 and 2-3",        4, {{0,1},{2,3}},                 false },
        { "tree plus one chord",        5, {{0,1},{1,2},{2,3},{3,4},{4,1}}, true }
    };
    cout << "graph                        truth   broken   repaired\n";
    int brokenWrong = 0, repairedWrong = 0;
    for (const Case& c : cases) {
        Graph G(c.n);
        for (auto [u, v] : c.es) G.addEdge(u, v);
        bool b = anyCycle(G, false), r = anyCycle(G, true);
        if (b != c.truth) ++brokenWrong;
        if (r != c.truth) ++repairedWrong;
        cout << "  " << c.name;
        for (size_t k = c.name.size(); k < 28; ++k) cout << ' ';
        cout << (c.truth ? "cycle  " : "none   ")
             << (b ? "cycle  " : "none   ") << "  "
             << (r ? "cycle" : "none")
             << (r == c.truth ? "" : "   <-- WRONG") << "\n";
    }
    cout << "broken version wrong on   " << brokenWrong << " of " << cases.size() << "
graphs\n";
    cout << "repaired version wrong on " << repairedWrong << " of " << cases.size() << "
graphs\n";
    return 0;
}
```

Its output:

```
graph                        truth   broken   repaired
  single edge 0-1             none   cycle    none
  path 0-1-2-3                none   cycle    none
  triangle 0-1-2-0            cycle  cycle    cycle
  star, center 0              none   cycle    none
  two parallel edges 0=1      cycle  cycle    cycle
  forest: 0-1 and 2-3         none   cycle    none
  tree plus one chord         cycle  cycle    cycle
broken version wrong on   4 of 7 graphs
repaired version wrong on 0 of 7 graphs
```

The broken version says “cycle” on every graph that has an edge at all, so it is right only by accident on the three graphs that really do have cycles.

11.4 Suppose every vertex had in-degree ≥ 1. Start anywhere and walk *backward* along an incoming edge; you can always continue, so the walk is infinite, and since V is finite some vertex repeats — giving a cycle. Contradiction. The out-degree statement is the same argument walking

forward. The in-degree-0 result is used implicitly in §11.3.11 at the start: it guarantees the ready set R is non-empty at line 2, so Kahn's loop actually begins.

11.5 From h: extract h (key 0); then g via (h,g)=1; f via (g,f)=2; c via (f,c)=4; i via (c,i)=2; d via (c,d)=7; now a and b both have key 8 (from h and from c), and the tie goes to a via (h,a)=8; extracting a lowers b's key to 4, so b follows via (a,b)=4; last e via (d,e)=9. The extraction order is h,g,f,c,i,d,a,b,e. Total $1+2+4+2+7+8+4+9=37$. The tree is $\{(g,h),(f,g),(c,f),(c,i),(a,h),(a,b),(c,d),(d,e)\}$ — **Kruskal's** tree, not Prim-from-a's. The weight is the same, as Theorem 11.5 guarantees; the tree depends on the root because the root changes which cuts the algorithm uses.

11.6 At the moment (g,i) is considered, the component is $\{c,f,g,h,i\}$ and the cycle created would be $g-i-c-f-g$, using (g,i)=6, (i,c)=2, (c,f)=4, (f,g)=2. Every other edge is 2, 4, 2 — all lighter than 6, which is exactly what Kruskal's sorted order guarantees: any edge already accepted was examined earlier, hence weighs no more.

11.7 Let G be connected on n vertices with $n-1$ edges, and suppose it has a cycle. Delete one edge of that cycle: the graph stays connected (the rest of the cycle supplies the missing route) and now has $n-2$ edges. Repeat until no cycle remains; the result is connected and acyclic, hence a tree, hence has $n-1$ edges by Theorem 11.3. But we deleted at least one edge, so it has at most $n-2$. Contradiction. So G is acyclic, and being connected and acyclic it is a tree. ■

11.8 (i) MST: take $a-b$ (1) and $b-c$ (1), weight 2. (ii) Shortest-path tree from a: $a-b$ costs 1; $a-c$ directly costs 1.5, versus 2 via b — so the tree is $\{a-b,\ a-c\}$, weight 2.5. Different trees. The line that changes is Prim's line 09: replacing $w(u,v) < key[v]$ by $key[u]+w(u,v) < key[v]$ turns it into Dijkstra.

11.9 With all three weights 1, each of the three spanning trees (drop one edge) has weight 2, so all three are MSTs. The criterion agrees: removing any tree edge still leaves an MST of weight 2. For the second part, this is Theorem 11.8: if $T_1 \neq T_2$ were both minimum, taking the lightest edge e in the symmetric difference and swapping it into the other tree strictly improves that tree, which is impossible. Distinctness is what makes the inequality $w(e) < w(f)$ strict.

11.10 *A simple method.* Compute an MST T once. For each of the $|V|-1$ edges $e \in T$, recompute the MST of $G-e$, and keep the lightest result. With the edges sorted once and reused, each recomputation is a Kruskal scan costing $O(E\,\alpha(V))$, so the whole method costs $O(E\lg E + VE\,\alpha(V))$ — within a factor $\alpha(V) \leq 4$ of the target.

An $O(VE)$ method. For every vertex u, walk the tree T from u by one depth-first search and record, for every v, $\max[u][v]$, the heaviest edge on the tree path from u to v; that is $O(V)$ per u and $O(V^2)$ in all. Adding a non-tree edge (u,v) to T closes exactly one cycle (§11.4.2), and the cheapest way to break it again is to drop the heaviest other edge on that cycle, $\max[u][v]$. So the second-best weight is $w(T) + \min_{(u,v)\notin T}(w(u,v) - w(\max[u][v]))$, found in $O(E)$. The total is $O(E\lg E + V^2)$, which is $O(VE)$ because $|E| \geq |V|-1$.

The swap claim. Let T' be a second-best tree. By the exchange argument of §11.7.3 applied repeatedly, T' can be turned into T by single-edge swaps, none of which increases the weight. Look at the last tree before T in that sequence: it differs from T by one swap, it weighs at most $w(T')$, and it is not T, so by the definition of second-best it weighs at least $w(T')$. It is therefore itself a second-best tree, one swap away from T.

11.11 Let T be an MST and T' any spanning tree with bottleneck $\beta' < \beta$, the bottleneck of T. Let $e \in T$ be an edge of weight β. Removing e splits T into two components, defining a cut. T' is

connected, so it has an edge f crossing that cut, and $w(f) \leq \beta' < \beta = w(e)$. Then $(T \setminus \{e\}) \cup \{f\}$ is a spanning tree of strictly smaller weight — contradiction. So no spanning tree has a smaller bottleneck than an MST. The converse is **false**: a minimum-bottleneck spanning tree need not be an MST. Take the triangle $a - b = 1$, $b - c = 5$, $a - c = 5$: the trees $\{ab, bc\}$ and $\{ab, ac\}$ both have bottleneck 5 and weight 6, while $\{bc, ac\}$ has bottleneck 5 and weight 10 — minimum bottleneck, not minimum weight.

11.12 *Dual of Theorem 11.5:* for any cut, a **maximum**-weight crossing edge belongs to some maximum spanning tree. The proof is the same exchange with the inequality reversed: the crossing edge (x, y) on the tree path satisfies $w(x, y) \leq w(u, v)$, so swapping does not decrease the weight. In **Listing 11.5**, change `return w < o.w;` to `return w > o.w;`. On the example graph the maximum spanning tree is $\{(d, f)14, (b, h)11, (e, f)10, (a, h)8, (b, c)8, (c, d)7, (h, i)7, (g, i)6\}$, weight $14 + 11 + 10 + 8 + 8 + 7 + 7 + 6 = 71$. (Equivalently: negate every weight and run ordinary Kruskal.)

11.13 Keep `heap` as an array of vertices and `pos[v]` as the index of v in it; every swap inside `siftUp`/`siftDown` updates both entries of `pos`. Then `decreaseKey(v, k)` is `key[v] = k; siftUp(pos[v]);`, which is $O(\lg V)$. The heap now holds at most $|V|$ entries instead of $O(E)$, so space falls from $O(E)$ to $O(V)$, and the running time becomes exactly $|V| \cdot O(\lg V) + O(E) \cdot O(\lg V) = O(E \lg V)$ rather than $O(E \lg E)$. Since $\lg E < 2 \lg V$, the improvement is a constant factor in time and a genuine factor of $|E|/|V|$ in space.

11.14 Replace the comparison sort by a counting sort on the weights: allocate $W + 1$ buckets, drop each edge into bucket $w(u, v)$, and concatenate. That costs $\Theta(E + W) = \Theta(E)$ when $W = O(E)$. The rest of the algorithm is unchanged, so the total becomes $\Theta(E) + O(E\,\alpha(V)) = O(E\,\alpha(V))$. **Nothing changes in the correctness proof**: it requires only that edges be processed in non-decreasing weight order, and a counting sort produces exactly that order.

11.15 Let T be any spanning tree. It has exactly $|V| - 1$ edges by Theorem 11.3, so $w'(T) = w(T) + (|V| - 1)c$ — the same constant is added to *every* spanning tree. Adding a constant to every element of a set does not change which element is smallest, so the minimizers coincide. For shortest paths the argument fails because different paths have different numbers of edges. Take $a - b = 1$, $b - c = 1$, $a - c = 3$. Shortest a to c is $a - b - c$, cost 2. Now add $c = 2$ to every weight: $a - b = 3$, $b - c = 3$, $a - c = 5$, and the shortest a to c is the direct edge, cost 5 against 6. The path changed.

11.16 (a) **False as stated, true if the lightest edge is unique.** If two edges tie for lightest, each is in some MST but perhaps not both in all: the all-1 triangle of Exercise 11.9 has three MSTs, and each omits one of the three lightest edges. If the minimum is unique, take $S = \{u\}$ for one endpoint: the edge is light for that cut, and by the strict version of the exchange argument it is in every MST. (b) **False.** In a tree, every edge — including the heaviest — is in the unique MST, because there is only one spanning tree. (c) **True.** Uniqueness makes the inequality in Step 5 of §11.7.3 strict, so any MST omitting the edge could be strictly improved, which is impossible.

11.17 In the code: nothing. Kruskal already accepts exactly the edges that join distinct components, so on a disconnected graph it terminates with a forest of k trees and $|V| - k$ edges, where k is the number of connected components. The only change is the *check*: assert $|A| = |V| - k$ rather than $|V| - 1$, and stop early after $|V| - k$ accepts rather than $|V| - 1$. In the proof of §11.10.11, the last paragraph must be re-read: "if two components remained, some edge of G would join them" is true only within a connected component of G, so the conclusion becomes "A

restricted to each component of G is a spanning tree of that component, and is minimum by the same safe-edge argument applied within it".

11.18 *Union by rank alone:* by Lemma 11.7, a rank-k root owns at least 2^k elements, so no rank exceeds $\lg n$ and no tree is taller than $\lg n$; every `FindSet` therefore walks at most $\lg n$ links. That is a **worst-case** bound and it is tight — the balanced binomial-style tree really can be that tall. *Path compression alone:* each find flattens the path it walked, so no node can be expensive twice in a row; averaged over a sequence, the cost is $O(\lg n)$ amortized, but a single find can still be $\Theta(n)$ because nothing stops unions from building a long chain in the first place. *Together:* union by rank keeps trees from ever *becoming* tall, and path compression destroys whatever height does appear, the moment anybody pays to walk it. Neither mechanism alone prevents repeated expensive walks; together, each unit of height can only be paid for a bounded number of times, and the bookkeeping that makes that precise gives the $O(\alpha(n))$ amortized bound. **Chapter 14** proves it, with the potential method; the original proof is Tarjan's [9], and Fredman and Saks [11] showed the bound cannot be improved.

References

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

[2] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962.

[3] A. Cayley, "A theorem on trees," *Quarterly Journal of Pure and Applied Mathematics*, vol. 23, pp. 376–378, 1889.

[4] V. Jarník, "O jistém problému minimálním" ["About a certain minimal problem"], *Práce Moravské Přírodovědecké Společnosti*, vol. 6, no. 4, pp. 57–63, 1930.

[5] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, Nov. 1957.

[6] R. L. Graham and P. Hell, "On the history of the minimum spanning tree problem," *Annals of the History of Computing*, vol. 7, no. 1, pp. 43–57, Jan. 1985.

[7] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, Feb. 1956.

[8] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[9] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, Apr. 1975.

[10] R. E. Tarjan, *Data Structures and Network Algorithms*. Philadelphia, PA, USA: SIAM, 1983.

[11] M. L. Fredman and M. E. Saks, "The cell probe complexity of dynamic data structures," in *Proc. 21st Annual ACM Symposium on Theory of Computing (STOC)*, Seattle, WA, USA, 1989, pp. 345–354.

[12] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

Chapter 12
Evaluation and Discussion III: Greedy Algorithms and Graphs

This chapter pauses the forward march of the book and consolidates everything Chapters 9, 10 and 11 have built: greedy algorithms and the exchange argument, activity selection, Huffman codes, the two knapsacks and matroids; breadth-first and depth-first search with everything they compute; directed acyclic graphs and topological sort; and minimum spanning trees by Prim and by Kruskal, with the disjoint-set forest that makes Kruskal fast. It contains no new theory. Instead it works problems — one complete design project and twenty-three further problems, every one solved from nothing, with every queue printed, every timestamp stamped, every union recorded and every intermediate number written down. By the end you will have taken a real problem from a blank page to a finished, measured program in nine stages (problem, model, algorithm choice, pseudocode, complexity, hand proof, implementation, evaluation, conclusion). You will have disproved a greedy rule with a counterexample and proved another with an exchange argument, built a Huffman code merge by merge and drawn its tree, run the matroid greedy algorithm on a scheduling problem and checked it against every subset, and traced breadth-first search, depth-first search on both kinds of graph, Kahn's algorithm, Kruskal with its disjoint-set arrays and Prim with both an array and a binary heap. Each problem ends with a *check yourself* variant and its answer. Thirty quick drills, a set of exercises with full solutions, and twelve complete C++17 programs round the chapter off; every program compiles with `g++ -std=c++17 -O2 -Wall` and is quoted with the output it really printed.

12.1 Where We Stand

12.1.1 What Chapters 9, 10 and 11 built

Three chapters have given us a second toolkit, and it is worth listing what is in it before we start using it, because every problem in this chapter reaches for one of these tools, and naming the tool is half of the work.

From **Chapter 9** we have the *greedy* strategy: build a solution one irrevocable choice at a time, always taking the option that looks best right now. We have the uncomfortable fact that this often fails (the coin systems of §9.3), and the two ingredients that make it succeed — the **greedy-choice property** and **optimal substructure** (§9.5). We have proved both ingredients for activity selection with an *exchange argument* (§9.4), seen greed win on the fractional knapsack and lose on the 0-1 knapsack (§9.6), built Huffman's optimal prefix-free code (§9.7), and met the **matroid**, the structure that explains in one theorem why a whole family of greedy algorithms is correct (§9.8).

From **Chapter 10** we have graphs: the vocabulary (vertices, edges, degree, paths, cycles, components, bipartite graphs), three ways to store a graph and what each costs (§10.5), and the two fundamental searches. **Breadth-first search** explores in rings of increasing distance and computes shortest hop-counts (§10.7); **depth-first search** dives as deep as it can, stamps every vertex with a discovery time and a finishing time, and classifies every edge as tree, back, forward

or cross (§10.8). With them we can find components, detect cycles and test bipartiteness in linear time (§10.9).

From **Chapter 11** we have **directed acyclic graphs** and the DAG theorem (§11.2), **topological sort** by depth-first finishing times and by Kahn's in-degree algorithm (§11.3), and **minimum spanning trees**: the cut property that makes greed safe (§11.7), **Prim–Jarník**, which grows one tree (§11.9), and **Kruskal**, which merges a forest with the help of a disjoint-set structure tuned by union by rank and path compression (§11.10). All of it follows Cormen et al. [1].

12.1.2 A revision map

Table 12.1 lists every idea of Chapters 9 to 11 that a problem can ask about, the section that teaches it, the typical question, and the place in this chapter where a problem of exactly that kind is worked. Use it in two directions: before you start, to find your own gaps; afterward, when a drill in §12.7 catches you out, to find the page that fixes it.

Table 12.1 Every idea of Chapters 9 to 11, the section that teaches it, the kind of question it raises, and the place in this chapter where a problem of that kind is worked in full.

Idea	Taught in	Typical question	Worked here
What makes an algorithm greedy	§9.2	Define the greedy-choice property	§12.3.2
Greedy failure: making change	§9.3	Give a coin system where greed loses, and the amount	§12.3.2
Activity selection, four rules	§9.4.3	Which rules are wrong, and on what instance	§12.3.3
Exchange argument	§9.4.7	Prove the earliest-finish rule optimal	§12.3.3, §12.3.4
Optimal substructure	§9.4.4, §9.5.3	State and prove it for a given problem	§12.3.3, §12.3.6
Fractional vs 0-1 knapsack	§9.6	Why greed works for one and not the other	§12.3.6
Huffman construction and cost	§9.7	Build the tree, compute $B(T)$	§12.3.5
Matroids, unit-time scheduling	§9.8	Independence, the exchange property, the greedy schedule	§12.3.7
Graph vocabulary and storage	§10.4, §10.5	Degree, path, cycle; list vs matrix cost	§12.4.2
Breadth-first search	§10.7	Trace it; prove $d[v] = \delta(s, v)$	§12.2.8, §12.4.2
Depth-first search, timestamps	§10.8	Trace it; the parenthesis theorem	§12.4.3, §12.4.4
Edge classification	§10.8.10	Classify every arc; which classes an undirected graph can have	§12.4.3, §12.4.4
Components, cycles, bipartiteness	§10.9	Two-color a graph; name the odd cycle	§12.4.5–§12.4.7
DAGs and the DAG theorem	§11.2	Prove: acyclic iff DFS finds no back edge	§12.5.2
Topological sort, both algorithms	§11.3	Produce an order by each; say why they differ	§12.2.8, §12.5.3, §12.5.4
Spanning trees, $\lvert V\rvert - 1$ edges	§11.4	Count the edges; count the trees	§12.2.5, §12.4.5
The cut property	§11.7	State it, prove it, apply it	§12.6.5
Prim–Jarník	§11.9	Trace with the priority queue	§12.6.3, §12.6.4
Kruskal and the disjoint-set forest	§11.10	Trace with `Find` and `Union`	§12.2.8, §12.6.2
Union by rank, path compression	§11.10.5–7	Draw the forest after each operation	§12.6.6
Uniqueness of the MST	§11.12	Is it unique? Produce a second one	§12.6.7

If one figure carries the whole of Chapter 11, it is the nine-vertex weighted graph used for every minimum-spanning-tree trace there, reproduced as **Figure 12.1**. Its answer — a tree of weight 37 — is worth committing to memory, because it is a yardstick against which you can check any Prim or Kruskal implementation in thirty seconds. Worked Problem T in §12.6.4 runs Prim on it again, this time with a binary heap, and lands on 37 from two different starting vertices.

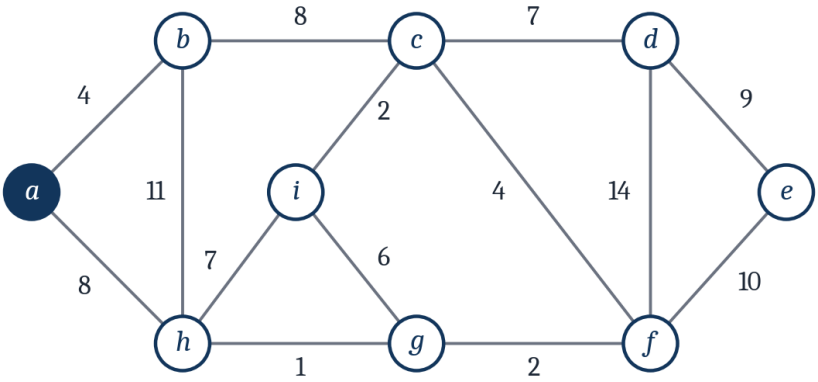


Figure 12.1 The nine-vertex, fourteen-edge weighted graph used for every minimum-spanning-tree trace in Chapter 11. Its minimum spanning tree weighs 37, and two different trees of that weight exist — which is why §11.12 exists.

12.1.3 How this chapter is organized

The chapter has four layers.

1. **One design project, worked end to end** (Section 12.2). A real problem for Profitina Laundry, the teaching case study of this book, taken through nine stages: the problem stated, the data modeled as graphs, the algorithms chosen against named alternatives, the pseudocode, the complexity predicted in advance, the design proved by hand on the real data, the implementation, the evaluation, and the conclusion. It uses breadth-first search, Kruskal's algorithm with a disjoint-set forest, and Kahn's topological sort on one data set, and it shows what "applying an algorithm" means when nobody has told you which algorithm applies.
2. **Twenty-three further worked problems** (Sections 12.3 to 12.6), grouped by chapter: six on greedy algorithms (A to F), six on searching a graph (G to L), five on DAGs and topological sort (M to Q) and six on minimum spanning trees (R to W). Each is set on data this book has not used before, so that working it is practice rather than recall, and each group ends with one complete program that computes every answer in the group.
3. **Thirty quick self-check drills** (Section 12.7), one line each, covering every idea in **Table 12.1**.
4. **The usual close**: where the same ideas appear in a real product (Section 12.8), the common mistakes (Section 12.9), a summary (Section 12.10), fifteen exercises (Section 12.11) and their full solutions (Section 12.12).

> **HOW TO USE THIS CHAPTER.** Read each problem statement, then cover the solution with a sheet of paper and try it yourself for ten minutes before reading on. A solution you have struggled with for ten minutes is understood in a way that a solution you have only read never is. The *check yourself* variant at the end of every problem exists so that you can test that understanding on something you have not seen worked; its answer follows immediately, so cover that too.

12.2 A Design Project, Worked End to End

12.2.1 What a design project is, and its nine stages

A textbook problem tells you what to compute. A real problem does not. It arrives as a business question — *which zones can we serve, which roads should we run, how fast can a bag be turned round?* — and the first and hardest step is to **recognize** it as an instance of something you have been taught. That skill is rarely exercised by short problems, because a short problem has already

done the recognizing for you. This section exercises it on one problem, from a blank page to a measured program.

The work falls into nine stages, and each one answers a single question. **Table 12.2** lists them; the subsections that follow work them in order.

Table 12.2 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

Two features of this list deserve a sentence each. First, **six of the nine stages happen before any code is written**. That is not bureaucracy: a design argued before the program exists is an argument for what the program *should* do, whereas a design written afterward merely describes what it *happens* to do, and cannot catch a mistake. Second, **stage 8 is not optional**. A program that has never been run against anything but its own example input has not been shown to work; it has only been shown to run.

12.2.2 Choosing a problem that deserves an algorithm

Before the nine stages begin, there is a choice: *is this problem worth an algorithm at all?* The overwhelmingly common failure in a first design project is an algorithm that is *decorative*. A pleasant little game calls breadth-first search on a 5×5 grid to move one sprite. The algorithm is present; it is not *load-bearing*. On a 5×5 grid any method works, so the design stage has nothing to argue and the evaluation stage has nothing to measure.

The test to apply to any idea is one sentence long.

> **THE LOAD-BEARING TEST.** *If the input grew by a factor of a thousand, would the naive method still work?* If yes, the algorithm is decoration and the design will have nothing to say. If no, you have a real problem, and the design writes itself: here is the naive method, here is why it dies, here is the algorithm that does not.

Table 12.3 applies the test to six ideas of the kind people actually propose when asked to build something with a graph algorithm.

Table 12.3 Six candidate projects put through the load-bearing test. Four survive it. The two that do not are not bad programs; they are poor *design* problems, because they give the design and the evaluation nothing to argue about.

Idea	Algorithm	Naive method	Does it survive ×1000?	Verdict
Sprite walks a 5 × 5 maze	BFS	try every path	yes, trivially	decoration
Pickup route over a city's roads	BFS, MST	try every subset of roads	no — 2^E	**load-bearing**
Minesweeper "reveal blank region"	flood fill (DFS/BFS)	re-scan the board per click	borderline	weak
Course-prerequisite planner	topological sort	guess an order, check it	no — $n!$ orders	**load-bearing**
Cheapest fiber network for n offices	Kruskal or Prim	enumerate spanning trees	no — Cayley: n^{n-2}	**load-bearing**
Social network "who is in my circle"	components, BFS	repeated reachability queries	no — $O(V(V+E))$ per batch	**load-bearing**

The project worked in this section is the fifth row crossed with the second: **Rute Laundry**, a same-day pickup and delivery planner for Profitina Laundry. It is close to the most common idea of all — *an application that computes the best delivery route between two locations* — so that you can see exactly how much more there is to say about that idea than a first attempt usually says.

12.2.3 Stage 1: the problem, stated

Profitina Laundry operates one outlet in Sukolilo, near the ITS campus, and serves the surrounding *kelurahan* (urban wards) by motorbike courier. Three questions arrive on the same morning.

1. **Which zones can be promised a two-hop pickup**, and which cannot be reached at all?
2. If the company wants a **private shuttle network** so that any bag can travel from any served zone to any other, **which roads should it operate** — recognizing that every kilometer operated costs money every day, forever?
3. A bag then passes through a fixed set of processing stages with precedence constraints. **In what order may they be run, and what is the fastest a single bag can be turned round?**

The inputs are a list of zones, a list of roads with their lengths, a list of processing stages with their durations, and a list of precedences between stages. The outputs are a hop count per zone, a set of roads, an order of stages and a turnaround time.

12.2.4 Stage 2: the model

All three questions are graph questions, and one data set answers all three.

The road network is an **undirected graph** $G = (V, E)$: the vertices are the fourteen zones, and $\{u, v\} \in E$ when a courier can ride directly between them. For question 1 the edges are unweighted, because every ride costs roughly the same time. For question 2 each edge carries a weight $w(u, v)$, the road distance in hundreds of meters. **Table 12.4** is the data.

Table 12.4 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		

The processing pipeline is a **directed graph**: the vertices are the ten stages, and an arc $x \to y$ means *x must finish before y may start*. It must be acyclic, or the specification is impossible to satisfy — which is itself something the program should check rather than assume. **Figure 12.2** shows what acyclic means — including a case that looks like a loop and is not.

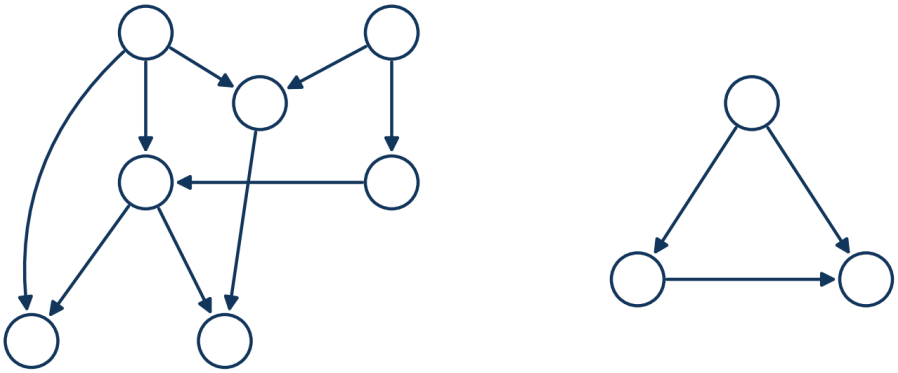


Figure 12.2 Two directed acyclic graphs. On the left, follow the arrows from anywhere and you run out of arrows. On the right, three arrows form a triangle that is still acyclic, because they do not run round it in one direction; reverse the bottom arrow and they would close a directed cycle, and a pipeline specification shaped like that could not be executed in any order at all.

12.2.5 Stage 3: the algorithms chosen, and the alternatives rejected

Question 1: breadth-first search from the outlet. The graph is unweighted, so the number of hops from the outlet to a zone is exactly the breadth-first distance $d[v]$, and BFS computes all of them in one pass at $\Theta(V + E)$ (§10.7.7). Any zone left with $d[v] = \infty$ is in a different connected component; that is not a scheduling problem but a second-outlet problem.

Rejected: Dijkstra's algorithm (Chapter 13). It is correct here, but it costs $O((V + E)\lg V)$ to compute something BFS gets in linear time, because with unit weights its priority queue degenerates into a plain queue. Use the weaker tool when the weaker tool suffices.

Also rejected: computing distances between all pairs of zones, at $\Theta(V^3)$ by the Floyd–Warshall method. We have one source, not V of them.

Question 2: Kruskal with a disjoint-set forest. Kruskal's algorithm [2] is the answer. We want a connected, acyclic, spanning subgraph of minimum total weight — the definition of a minimum spanning tree (§11.5). **Figure 12.3** shows what "spanning tree" means.

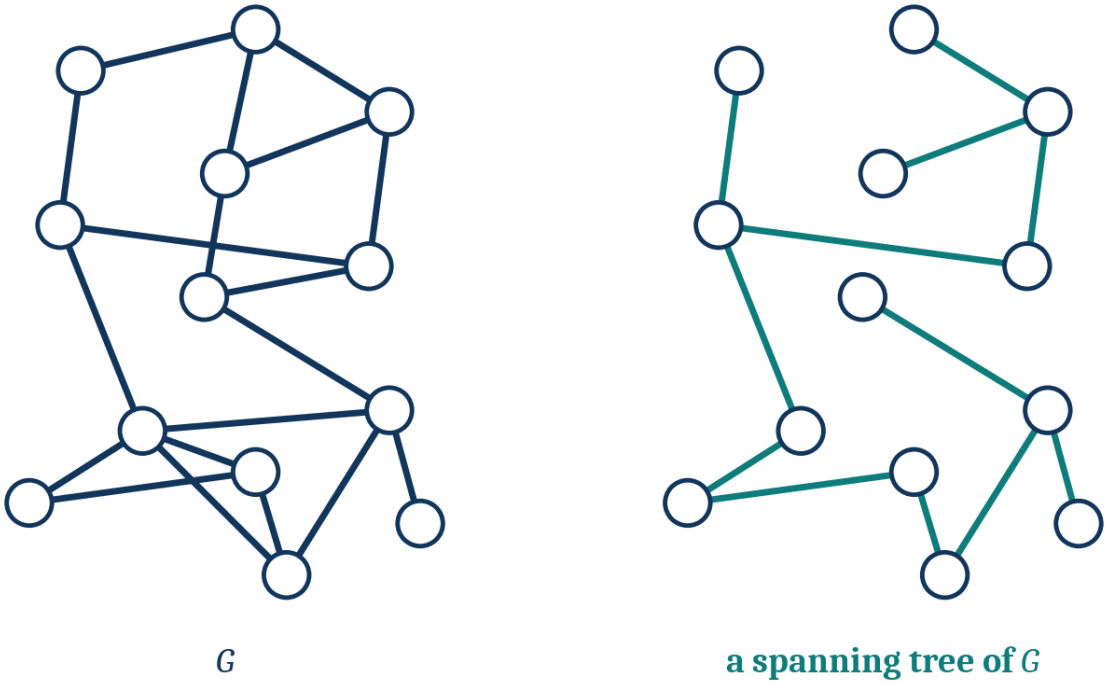


Figure 12.3 A spanning tree. **Left:** a graph. **Right:** a spanning tree of it — every vertex present, still connected, no cycle; thirteen vertices and twelve edges, which is the $|V| - 1$ rule of §11.4.2. The shuttle backbone of question 2 is exactly this object, chosen to be as light as possible.

Rejected: enumerate the spanning trees and pick the lightest. The number of spanning trees of the complete graph K_n on n vertices is given by Cayley's formula,

$$\tau(K_n) = n^{n-2}, \tag{12.1}$$

so by Equation (12.1) a complete graph on twelve vertices already has $12^{10} = 61\,917\,364\,224$ spanning trees. This is the load-bearing test of §12.2.2 passed with room to spare.

Rejected: Prim–Jarník [3]. It is equally correct and gives a tree of the same weight (§11.11). Kruskal is chosen because the data arrives as an **edge list** — the operations team recorded one row per road driven — and Kruskal consumes an edge list directly, whereas Prim must first build an adjacency structure. On a sparse graph the two costs, $O(E\lg E)$ and $O(E\lg V)$, are within a constant factor of each other, so the deciding factor is the input format, and that is the argument to write down.

Question 3: Kahn's in-degree topological sort [4]. A legal work order is exactly a topological order of the precedence DAG (§11.3). Kahn is chosen over the depth-first method because it detects a cyclic specification *for free*: if it emits fewer than $|V|$ stages, a cycle exists, and the stages it failed to emit are precisely those on or downstream of the cycle (§11.3.13). The depth-first method needs a separate back-edge test to say the same thing.

The same pass also gives the **critical path**: because the vertices come out in topological order, one relaxation per arc computes the longest chain, and that chain is the minimum turnaround time. Writing $dur[v]$ for the duration of stage v and $ef[v]$ for its earliest finishing time,

$$ef[v] = dur[v] + \max(\{0\} \cup \{\, ef[u] : u \to v \,\}). \tag{12.2}$$

Equation (12.2) says: a stage can finish no earlier than its own duration after the latest of its prerequisites has finished, and a stage with no prerequisite finishes after its own duration alone.

Because Kahn emits every u before any v with $u \to v$, every $ef[u]$ on the right is final by the time v needs it.

12.2.6 Stage 4: the pseudocode

Pseudocode 12.1 is the whole design in eleven lines. Every numbered line will have a home in the program of §12.2.9.

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)
```

12.2.7 Stage 5: the complexity, worked out in advance

Let $n = |V|$ zones, $m = |E|$ roads, s stages and p precedences. **Table 12.5** is the prediction — written before the program exists, so that the evaluation has something to test against.

Table 12.5 The predicted cost of Rute Laundry, phase by phase, written before any code was compiled. The sorting step dominates, and §12.2.10 measures whether that is what actually happens.

Phase	Cost	Why
Build adjacency lists	$\Theta(n+m)$	one pass over the edge list
BFS from the outlet	$\Theta(n+m)$	each vertex enqueued once, each edge scanned twice (§10.7.7)
Connected components	$\Theta(n+m)$	one BFS per component, disjoint edge sets
Sort the roads	$\Theta(m\lg m)$	comparison sort; this dominates
m `Find`s and $n-1$ `Union`s	$O(m\,\alpha(n))$	union by rank with path compression (§11.10.7)
Kahn plus the critical path	$\Theta(s+p)$	each arc relaxed once
Total	$\Theta(m\lg m + n + s + p)$	

Adding the rows gives the total

$$T(n,m,s,p) = \Theta(m\lg m + n + s + p), \tag{12.3}$$

where α in the fifth row is the inverse Ackermann function of §11.10.7, which is at most 4 for any n that will ever be run on physical hardware. So the disjoint-set work is effectively linear, and on a connected road network (where $m \geq n-1$) Equation (12.3) is simply $\Theta(m\lg m)$ plus the pipeline: the sort is genuinely the bottleneck.

12.2.8 Stage 6: the design proved by hand, before any code

This is the stage most often skipped and most worth doing. **Work the algorithm by hand on the real data, and keep the trace.** If the hand trace and the program later disagree, one of them is wrong, and you will find out in an afternoon instead of the night before the program is needed.

Question 1 by hand. Breadth-first search from Sukolilo, adjacency lists in the order the roads were recorded. Initialize $d[\text{Sukolilo}] = 0$ and everything else ∞; enqueue Sukolilo.

- **Dequeue Sukolilo.** Neighbors Menur and Keputih are white: $d = 1$ each. Queue: (Menur:1, Keputih:1).
- **Dequeue Menur.** Sukolilo is black. Gebang and Semolowaru are white: $d = 2$. Queue: (Keputih:1, Gebang:2, Semolowaru:2).
- **Dequeue Keputih.** Sukolilo black, Gebang already gray — *skip it; first touch wins.* Kejawan white: $d = 2$. Queue: (Gebang:2, Semolowaru:2, Kejawan:2).
- **Dequeue Gebang.** Klampis white: $d = 3$. Queue: (Semolowaru:2, Kejawan:2, Klampis:3).
- **Dequeue Semolowaru.** Nginden white: $d = 3$; Klampis already gray. Queue: (Kejawan:2, Klampis:3, Nginden:3).
- **Dequeue Kejawan.** Mulyosari white: $d = 3$. Queue: (Klampis:3, Nginden:3, Mulyosari:3).
- **Dequeue Klampis.** Manyar white: $d = 4$; Mulyosari gray. Queue: (Nginden:3, Mulyosari:3, Manyar:4).
- **Dequeue Nginden.** Manyar gray. Nothing new.
- **Dequeue Mulyosari.** Kalijudan white: $d = 4$. Queue: (Manyar:4, Kalijudan:4).
- **Dequeue Manyar.** DukuhSetro white: $d = 5$. Queue: (Kalijudan:4, DukuhSetro:5).
- **Dequeue Kalijudan.** DukuhSetro gray. Nothing new.
- **Dequeue DukuhSetro.** All neighbors black. Queue empty; the loop ends.

Twelve vertices were dequeued, and Bulak and Kenjeran were never touched at all. **Table 12.6** collects the answer.

Table 12.6 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

Question 2 by hand. Sort the sixteen roads of the main component by weight and run Kruskal, breaking ties by the order of the endpoints in the zone list. **Table 12.7** is the whole run.

Table 12.7 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
13–16	Gebang–Klampis 10, Semolowaru–Klampis 11, Keputih–Kejawan 12, Klampis–Mulyosari 13		all reject	69

Two things deserve attention. First, the rejection at step 9 has nothing to do with the weight 8: the very next edge, also of weight 8, is accepted. The decision depends only on the partition, that is, on which zones are already joined. Second, running all sixteen roads would cost $8 + 6 + 7 + 9 + 5 + 12 + 10 + 4 + 11 + 3 + 13 + 5 + 8 + 9 + 6 + 7 = 123$, so the backbone saves $123 - 69 = 54$, which is $54/123 = 43.9\%$ of the daily distance. That percentage is the sentence that belongs in the conclusion.

Question 3 by hand. The ten stages, with durations in minutes, are intake 2, weigh 3, sort 5, pretreat 8, wash 30, dry 45, press 12, fold 10, pack 6 and invoice 4; the precedences are intake→weigh, weigh→sort, weigh→invoice, sort→pretreat, sort→wash, pretreat→wash, wash→dry, dry→press, press→fold, fold→pack and invoice→pack. The in-degrees start at intake 0, weigh 1, sort 1, pretreat 1, wash 2, dry 1, press 1, fold 1, pack 2, invoice 1, so only *intake* has in-degree zero and the queue begins with it alone. **Table 12.8** is the run.

Table 12.8 Kahn's algorithm on the ten pipeline stages. Ten stages emitted in ten rounds, so the specification is acyclic and the order is legal. In round 4 *invoice* is emitted before *pretreat*, not because it is urgent but because the queue is first-in-first-out and invoice was enqueued first.

Round	Emit	Its arcs reduce	Newly at zero, enqueued
1	intake	weigh 1 → 0	weigh
2	weigh	sort 1 → 0, invoice 1 → 0	sort, invoice
3	sort	pretreat 1 → 0, wash 2 → 1	pretreat
4	invoice	pack 2 → 1	—
5	pretreat	wash 1 → 0	wash
6	wash	dry 1 → 0	dry
7	dry	press 1 → 0	press
8	press	fold 1 → 0	fold
9	fold	pack 1 → 0	pack
10	pack	—	—

Applying Equation (12.2) as each stage is emitted gives the earliest finishing time of every stage: intake 2; weigh $2 + 3 = 5$; sort $5 + 5 = 10$; invoice $5 + 4 = 9$; pretreat $10 + 8 = 18$; wash $\max(10,18) + 30 = 48$; dry $48 + 45 = 93$; press $93 + 12 = 105$; fold $105 + 10 = 115$; pack $\max(115,9) + 6 = 121$. The longest chain is

$$\text{intake} \to \text{weigh} \to \text{sort} \to \text{pretreat} \to \text{wash} \to \text{dry} \to \text{press} \to \text{fold} \to \text{pack},$$

costing $2 + 3 + 5 + 8 + 30 + 45 + 12 + 10 + 6 = 121$ minutes. *Invoice* is not on it, which is the useful business finding: raising the invoice can be delayed by up to $115 - 9 = 106$ minutes without delaying the bag.

12.2.9 Stage 7: the implementation

Now, and only now, the program. **Listing 12.1** is the complete implementation: it compiles with one command, runs with no arguments, and prints an answer to each of the three questions. Read it against **Pseudocode 12.1** — every numbered line there has a home here.

Listing 12.1 Rute Laundry: breadth-first search and components, Kruskal with a disjoint-set forest, and Kahn's topological sort with the critical path. Compile with `g++ -std=c++17 -O2 -Wall -o router router.cpp` and run `./router`.

```
// Rute Laundry -- a design project for Profitina Laundry, worked end to end.
// Three algorithms from Chapters 10 and 11, on one working day's data:
//   Part A  breadth-first search over the road network: how many hops is
//           each zone from the outlet, and which zones no courier reaches?
//   Part B  Kruskal with a disjoint-set forest: the cheapest set of shuttle
//           links that keeps every reachable zone connected.
//   Part C  Kahn's topological sort of the job pipeline, plus the critical
//           path -- the fastest a single bag can possibly be turned round.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
using namespace std;

// ------------------------------------------------ the zone road network
static const vector<string> ZONE = {
    "Sukolilo", "Menur", "Keputih", "Gebang", "Semolowaru", "Nginden",
    "Klampis", "Manyar", "Mulyosari", "Kalijudan", "Kejawan", "DukuhSetro",
    "Bulak", "Kenjeran"
};
struct Road { int a, b, hm; };                  // hm = distance in hundreds of meters
static const vector<Road> ROAD = {
    {0,1,8}, {0,2,6}, {1,3,7}, {1,4,9}, {2,3,5}, {2,10,12}, {3,6,10},
    {4,5,4}, {4,6,11}, {5,7,6}, {6,7,3}, {6,8,13}, {8,9,5}, {9,11,8},
    {7,11,9}, {10,8,7}, {12,13,4}
};

// ------------------------------------------------------ Part A: BFS
struct Bfs { vector<int> d, pi; };

Bfs bfs(const vector<vector<int>>& adj, int s) {
    Bfs R;
    R.d.assign(adj.size(), INT_MAX);
    R.pi.assign(adj.size(), -1);
    R.d[s] = 0;
    queue<int> Q; Q.push(s);
    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        for (int v : adj[u])
            if (R.d[v] == INT_MAX) { R.d[v] = R.d[u] + 1; R.pi[v] = u; Q.push(v); }
    }
    return R;
}
```

Listing 12.1 (Continued) *Rute Laundry: breadth-first search and components, Kruskal with a disjoint-set forest, and Kahn's topological sort with the critical path. Compile with g++ -std=c++17 -O2 -Wall -o router router.cpp and run ./router.*

```
string routeTo(const Bfs& R, int v) {                    // walk pi back, then reverse
    if (R.d[v] == INT_MAX) return "-- no route --";
    vector<int> p;
    for (int x = v; x != -1; x = R.pi[x]) p.push_back(x);
    reverse(p.begin(), p.end());
    string s;
    for (size_t i = 0; i < p.size(); ++i) { if (i) s += " -> "; s += ZONE[p[i]]; }
    return s;
}

// ------------------------------------------ Part B: disjoint-set forest
struct DSU {
    vector<int> parent, rnk;
    explicit DSU(int n) : parent(n), rnk(n, 0) {
        for (int i = 0; i < n; ++i) parent[i] = i;
    }
    int find(int x) { return parent[x] == x ? x : parent[x] = find(parent[x]); }
    bool unite(int x, int y) {                            // union by rank
        x = find(x); y = find(y);
        if (x == y) return false;
        if (rnk[x] < rnk[y]) swap(x, y);
        parent[y] = x;
        if (rnk[x] == rnk[y]) ++rnk[x];
        return true;
    }
};

// ---------------------------------------------- Part C: the pipeline DAG
static const vector<string> STAGE = {
    "intake", "weigh", "sort", "pretreat", "wash",
    "dry", "press", "fold", "pack", "invoice"
};
static const vector<int> MINUTES = { 2, 3, 5, 8, 30, 45, 12, 10, 6, 4 };
static const vector<pair<int,int>> PREC = {
    {0,1}, {1,2}, {1,9}, {2,3}, {2,4}, {3,4},
    {4,5}, {5,6}, {6,7}, {7,8}, {9,8}
};

int main() {
    const int n = (int)ZONE.size();
    vector<vector<int>> adj(n);
    for (const Road& r : ROAD) { adj[r.a].push_back(r.b); adj[r.b].push_back(r.a); }
```

Listing 12.1 (Continued) *Rute Laundry: breadth-first search and components, Kruskal with a disjoint-set forest, and Kahn's topological sort with the critical path. Compile with g++ -std=c++17 -O2 -Wall -o router router.cpp and run ./router.*

```
    cout << "=== Part A: service area, by breadth-first search from Sukolilo ===\n";
    Bfs R = bfs(adj, 0);
    cout << left << setw(12) << "zone" << right << setw(6) << "hops" << "   route\n";
    for (int v = 0; v < n; ++v) {
        cout << left << setw(12) << ZONE[v] << right << setw(6);
        if (R.d[v] == INT_MAX) cout << "inf"; else cout << R.d[v];
        cout << "   " << routeTo(R, v) << "\n";
    }
    int twoHop = 0;
    for (int v = 0; v < n; ++v) if (R.d[v] <= 2) ++twoHop;
    cout << "zones within the two-hop promise: " << twoHop << " of " << n << "\n";

    // connected components, by repeated BFS
    vector<int> comp(n, -1); int nc = 0;
    for (int v = 0; v < n; ++v) if (comp[v] == -1) {
        queue<int> Q; Q.push(v); comp[v] = nc;
        while (!Q.empty()) { int u = Q.front(); Q.pop();
            for (int w : adj[u]) if (comp[w] == -1) { comp[w] = nc; Q.push(w); } }
        ++nc;
    }
    cout << "connected components: " << nc << "\n";
    for (int c = 0; c < nc; ++c) {
        cout << "  component " << c << ":";
        for (int v = 0; v < n; ++v) if (comp[v] == c) cout << " " << ZONE[v];
        cout << "\n";
    }

    cout << "\n=== Part B: the shuttle backbone, by Kruskal ===\n";
    // only the component that contains the outlet can be served by one backbone
    vector<Road> cand;
    for (const Road& r : ROAD) if (comp[r.a] == 0) cand.push_back(r);
    sort(cand.begin(), cand.end(), [](const Road& x, const Road& y) {
        if (x.hm != y.hm) return x.hm < y.hm;
        if (x.a != y.a) return x.a < y.a;
        return x.b < y.b;
    });
    int inComp = 0; for (int v = 0; v < n; ++v) if (comp[v] == 0) ++inComp;
    DSU dsu(n);
    int total = 0, taken = 0, examined = 0;
    cout << setw(4) << "w" << "  edge                      verdict\n";
    for (const Road& r : cand) {
        ++examined;
        bool ok = dsu.unite(r.a, r.b);
        cout << setw(4) << r.hm << "  " << left << setw(24)
             << (ZONE[r.a] + "-" + ZONE[r.b]) << right
             << (ok ? "ACCEPT" : "reject (cycle)") << "\n";
```

Listing 12.1 (Continued) *Rute Laundry: breadth-first search and components, Kruskal with a disjoint-set forest, and Kahn's topological sort with the critical path. Compile with g++ -std=c++17 -O2 -Wall -o router router.cpp and run ./router.*

```
        if (ok) { total += r.hm; ++taken; }
    }
    cout << "edges examined " << examined << ", accepted " << taken
         << " (= " << inComp << " - 1), total " << total
         << " hm = " << fixed << setprecision(1) << total / 10.0 << " km\n";
    int all = 0; for (const Road& r : cand) all += r.hm;
    cout << "all candidate roads would cost " << all << " hm; the backbone saves "
         << all - total << " hm (" << setprecision(1)
         << 100.0 * (all - total) / all << "%)\n";

    cout << "\n=== Part C: the job pipeline, by Kahn ===\n";
    const int m = (int)STAGE.size();
    vector<vector<int>> dag(m); vector<int> indeg(m, 0);
    for (auto [u, v] : PREC) { dag[u].push_back(v); ++indeg[v]; }
    vector<int> ef(m, 0), pred(m, -1), order;
    queue<int> Q;
    for (int v = 0; v < m; ++v) if (indeg[v] == 0) Q.push(v);
    cout << "round  emit       in-degrees after the removal\n";
    int round = 0;
    while (!Q.empty()) {
        int u = Q.front(); Q.pop(); order.push_back(u);
        if (ef[u] == 0) ef[u] = MINUTES[u];                // a source stage
        for (int v : dag[u]) {
            if (ef[u] + MINUTES[v] > ef[v]) { ef[v] = ef[u] + MINUTES[v]; pred[v] = u; }
            if (--indeg[v] == 0) Q.push(v);
        }
        cout << setw(5) << ++round << "  " << left << setw(10) << STAGE[u] << right;
        for (int v = 0; v < m; ++v) cout << " " << indeg[v];
        cout << "\n";
    }
    cout << "emitted " << order.size() << " of " << m
         << (order.size() == (size_t)m ? "  -> acyclic, the order is legal\n"
                                       : "  -> a cycle exists\n");
    cout << "topological order:";
    for (int v : order) cout << " " << STAGE[v];
    cout << "\n";
    cout << "earliest finish, minute by minute:\n";
    for (int v : order)
        cout << "  " << left << setw(10) << STAGE[v] << right << setw(4) << MINUTES[v]
             << " min, done at " << setw(4) << ef[v] << "\n";
    int last = (int)(max_element(ef.begin(), ef.end()) - ef.begin());
    vector<int> crit;
    for (int x = last; x != -1; x = pred[x]) crit.push_back(x);
    reverse(crit.begin(), crit.end());
    cout << "critical path (" << ef[last] << " min):";
    for (size_t i = 0; i < crit.size(); ++i)
```

Listing 12.1 (Continued) *Rute Laundry: breadth-first search and components, Kruskal with a disjoint-set forest, and Kahn's topological sort with the critical path. Compile with g++ -std=c++17 -O2 -Wall -o router router.cpp and run ./router.*

```
        cout << (i ? " -> " : " ") << STAGE[crit[i]];
    cout << "\n";
    return 0;
}
```

Running it produces exactly this, first for question 1:

```
=== Part A: service area, by breadth-first search from Sukolilo ===
zone          hops   route
Sukolilo         0   Sukolilo
Menur            1   Sukolilo -> Menur
Keputih          1   Sukolilo -> Keputih
Gebang           2   Sukolilo -> Menur -> Gebang
Semolowaru       2   Sukolilo -> Menur -> Semolowaru
Nginden          3   Sukolilo -> Menur -> Semolowaru -> Nginden
Klampis          3   Sukolilo -> Menur -> Gebang -> Klampis
Manyar           4   Sukolilo -> Menur -> Gebang -> Klampis -> Manyar
Mulyosari        3   Sukolilo -> Keputih -> Kejawan -> Mulyosari
Kalijudan        4   Sukolilo -> Keputih -> Kejawan -> Mulyosari -> Kalijudan
Kejawan          2   Sukolilo -> Keputih -> Kejawan
DukuhSetro       5   Sukolilo -> Menur -> Gebang -> Klampis -> Manyar -> DukuhSetro
Bulak          inf   -- no route --
Kenjeran       inf   -- no route --
zones within the two-hop promise: 6 of 14
connected components: 2
  component 0: Sukolilo Menur Keputih Gebang Semolowaru Nginden Klampis Manyar Mulyosari
Kalijudan Kejawan DukuhSetro
  component 1: Bulak Kenjeran
```

Compare that against **Table 12.6**, computed by hand a page earlier: every distance agrees, including the two infinities. That agreement is the first line of the evaluation, and it cost nothing to obtain because the hand trace already existed.

The components result is the one to read twice. A graph with several components looks like **Figure 12.4**: pieces with no edge between them, and inside each piece every vertex reaching every other.

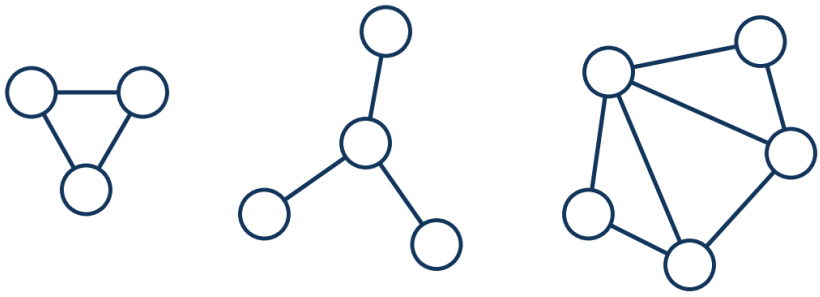


Figure 12.4 A graph with three connected components: a triangle, a star, and a five-vertex piece. No edge crosses between them. The Rute Laundry road network has two such pieces, and the second one — Bulak with Kenjeran — is the reason a two-hop promise must not be advertised for the whole city.

Part B prints the whole Kruskal run, and it matches **Table 12.7** line for line:

```
=== Part B: the shuttle backbone, by Kruskal ===
   w  edge                      verdict
   3  Klampis-Manyar          ACCEPT
   4  Semolowaru-Nginden      ACCEPT
   5  Keputih-Gebang          ACCEPT
   5  Mulyosari-Kalijudan     ACCEPT
   6  Sukolilo-Keputih        ACCEPT
   6  Nginden-Manyar          ACCEPT
   7  Menur-Gebang            ACCEPT
   7  Kejawan-Mulyosari       ACCEPT
   8  Sukolilo-Menur          reject (cycle)
   8  Kalijudan-DukuhSetro    ACCEPT
   9  Menur-Semolowaru        ACCEPT
   9  Manyar-DukuhSetro       ACCEPT
  10  Gebang-Klampis          reject (cycle)
  11  Semolowaru-Klampis      reject (cycle)
  12  Keputih-Kejawan         reject (cycle)
  13  Klampis-Mulyosari       reject (cycle)
edges examined 16, accepted 11 (= 12 - 1), total 69 hm = 6.9 km
all candidate roads would cost 123 hm; the backbone saves 54 hm (43.9%)
```

And Part C:

```
=== Part C: the job pipeline, by Kahn ===
round  emit       in-degrees after the removal
    1  intake     0 0 1 1 2 1 1 1 2 1
    2  weigh      0 0 0 1 2 1 1 1 2 0
    3  sort       0 0 0 0 1 1 1 1 2 0
    4  invoice    0 0 0 0 1 1 1 1 1 0
    5  pretreat   0 0 0 0 0 1 1 1 1 0
    6  wash       0 0 0 0 0 0 1 1 1 0
    7  dry        0 0 0 0 0 0 0 1 1 0
    8  press      0 0 0 0 0 0 0 0 1 0
    9  fold       0 0 0 0 0 0 0 0 0 0
   10  pack       0 0 0 0 0 0 0 0 0 0
emitted 10 of 10  -> acyclic, the order is legal
topological order: intake weigh sort invoice pretreat wash dry press fold pack
earliest finish, minute by minute:
  intake       2 min, done at    2
  weigh        3 min, done at    5
  sort         5 min, done at   10
  invoice      4 min, done at    9
  pretreat     8 min, done at   18
  wash        30 min, done at   48
  dry         45 min, done at   93
  press       12 min, done at  105
  fold        10 min, done at  115
  pack         6 min, done at  121
critical path (121 min): intake -> weigh -> sort -> pretreat -> wash -> dry -> press ->
fold -> pack
```

Three details in that listing are worth noticing.

`find` compresses on the way home. The line `return parent[x] == x ? x : parent[x] = find(parent[x]);` does two jobs at once: it finds the representative, and on the way back up the recursion it re-points every node it passed straight at the root. That single assignment is the whole of path compression (§11.10.6).

A rejected edge still changes the forest. At the `Sukolilo-Menur` rejection, two `find` calls ran before the comparison failed, and both compressed their paths. Someone watching only the accept/reject column would never see it.

The pipeline test is free. `emitted 10 of 10` is the acyclicity certificate. Nothing extra was computed to obtain it; it fell out of Kahn's loop. That is the argument §12.2.5 used to prefer Kahn over the depth-first method, now visible in the output.

12.2.10 Stage 8: the evaluation

An evaluation answers three questions, and no others: **is it right, does it scale as predicted, and what does it buy the business?** **Listing 12.2** answers the first two mechanically.

Listing 12.2 The evaluation harness: a cross-check of Kruskal against Prim, a postcondition check on BFS, and a timing run at five sizes with the elapsed time divided by the predicted cost. Compile with `g++ -std=c++17 -O2 -Wall -o routereval routereval.cpp` and run `./routereval`.

```
// The evaluation harness for the Rute Laundry design project.
// It answers the three questions every evaluation must answer:
//   1. is the implementation CORRECT?   -- Kruskal against Prim, 2000 graphs,
//                                          and every postcondition checked;
//   2. does it SCALE as the analysis predicts?  -- timed at five sizes;
//   3. what does it cost the BUSINESS?  -- the backbone against the full road set.
#include <iostream>
#include <iomanip>
#include <vector>
#include <queue>
#include <algorithm>
#include <random>
#include <chrono>
#include <climits>
#include <cmath>
using namespace std;
using Clock = chrono::steady_clock;

struct Edge { int a, b, w; };

// ------------------------------------------------------------ Kruskal
struct DSU {
    vector<int> p, r;
    explicit DSU(int n) : p(n), r(n, 0) { for (int i = 0; i < n; ++i) p[i] = i; }
    int find(int x) { return p[x] == x ? x : p[x] = find(p[x]); }
    bool unite(int x, int y) {
        x = find(x); y = find(y);
        if (x == y) return false;
        if (r[x] < r[y]) swap(x, y);
        p[y] = x; if (r[x] == r[y]) ++r[x];
        return true;
    }
};

long long kruskal(int n, vector<Edge> e) {
    sort(e.begin(), e.end(), [](const Edge& x, const Edge& y) { return x.w < y.w; });
    DSU d(n); long long t = 0; int k = 0;
    for (const Edge& x : e) if (d.unite(x.a, x.b)) { t += x.w; if (++k == n - 1) break; }
    return k == n - 1 ? t : -1;
}
```

Listing 12.2 (Continued) *The evaluation harness: a cross-check of Kruskal against Prim, a postcondition check on BFS, and a timing run at five sizes with the elapsed time divided by the predicted cost. Compile with g++ -std=c++17 -O2 -Wall -o routereval routereval.cpp and run ./routereval.*

```
// ---------------------------------------------------------------- Prim
long long prim(int n, const vector<Edge>& e) {
    vector<vector<pair<int,int>>> adj(n);
    for (const Edge& x : e) { adj[x.a].push_back({x.b, x.w}); adj[x.b].push_back({x.a,
x.w}); }
    vector<int> key(n, INT_MAX); vector<char> in(n, 1);
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    key[0] = 0; Q.push({0, 0});
    long long t = 0; int taken = 0;
    while (!Q.empty()) {
        auto [k, u] = Q.top(); Q.pop();
        if (!in[u] || k != key[u]) continue;                // a stale queue entry
        in[u] = 0; t += k; ++taken;
        for (auto [v, w] : adj[u]) if (in[v] && w < key[v]) { key[v] = w; Q.push({w, v}); }
    }
    return taken == n ? t : -1;
}

// ---------------------------------------------------------------- BFS
vector<int> bfs(const vector<vector<int>>& adj, int s) {
    vector<int> d(adj.size(), INT_MAX);
    d[s] = 0; queue<int> Q; Q.push(s);
    while (!Q.empty()) { int u = Q.front(); Q.pop();
        for (int v : adj[u]) if (d[v] == INT_MAX) { d[v] = d[u] + 1; Q.push(v); } }
    return d;
}

int main() {
    mt19937 rng(20260515);

    cout << "=== Test 1: Kruskal and Prim must agree, on 2000 random graphs ===\n";
    int mismatch = 0;
    for (int t = 0; t < 2000; ++t) {
        int n = 5 + (int)(rng() % 25);
        vector<Edge> e;
        for (int v = 1; v < n; ++v)                         // a random spanning tree
first
            e.push_back({(int)(rng() % v), v, 1 + (int)(rng() % 50)});
        int extra = (int)(rng() % (2 * n));
        for (int i = 0; i < extra; ++i) {
            int a = (int)(rng() % n), b = (int)(rng() % n);
            if (a != b) e.push_back({a, b, 1 + (int)(rng() % 50)});
        }
        if (kruskal(n, e) != prim(n, e)) ++mismatch;
    }
    cout << "mismatches: " << mismatch << " out of 2000\n";
```

Listing 12.2 (Continued) *The evaluation harness: a cross-check of Kruskal against Prim, a postcondition check on BFS, and a timing run at five sizes with the elapsed time divided by the predicted cost. Compile with g++ -std=c++17 -O2 -Wall -o routereval routereval.cpp and run ./routereval.*

```
    cout << "\n=== Test 2: the BFS postcondition, on 2000 random graphs ===\n";
    int broken = 0;
    for (int t = 0; t < 2000; ++t) {
        int n = 5 + (int)(rng() % 25);
        vector<vector<int>> adj(n);
        for (int v = 1; v < n; ++v) { int u = (int)(rng() % v);
            adj[u].push_back(v); adj[v].push_back(u); }
        int extra = (int)(rng() % (2 * n));
        for (int i = 0; i < extra; ++i) { int a = (int)(rng()%n), b = (int)(rng()%n);
            if (a != b) { adj[a].push_back(b); adj[b].push_back(a); } }
        vector<int> d = bfs(adj, 0);
        for (int u = 0; u < n; ++u) for (int v : adj[u])
            if (abs(d[u] - d[v]) > 1) ++broken;              // no edge may span two levels
    }
    cout << "edges spanning more than one BFS level: " << broken << "\n";

    cout << "\n=== Test 3: does it scale as the analysis says? ===\n";
    cout << setw(9) << "zones" << setw(10) << "roads"
         << setw(12) << "BFS ms" << setw(12) << "Kruskal ms"
         << setw(15) << "BFS/(V+E) us" << setw(18) << "Kruskal/(ElgE) us" << "\n";
    for (int n : {2000, 4000, 8000, 16000, 32000}) {
        int m = 4 * n;
        vector<Edge> e; vector<vector<int>> adj(n);
        for (int v = 1; v < n; ++v) {
            int u = (int)(rng() % v), w = 1 + (int)(rng() % 1000);
            e.push_back({u, v, w}); adj[u].push_back(v); adj[v].push_back(u);
        }
        while ((int)e.size() < m) {
            int a = (int)(rng()%n), b = (int)(rng()%n);
            if (a == b) continue;
            int w = 1 + (int)(rng() % 1000);
            e.push_back({a, b, w}); adj[a].push_back(b); adj[b].push_back(a);
        }
        auto t0 = Clock::now(); volatile int sink = bfs(adj, 0)[n-1]; (void)sink;
        auto t1 = Clock::now(); volatile long long s2 = kruskal(n, e); (void)s2;
        auto t2 = Clock::now();
        double bms = chrono::duration<double, milli>(t1 - t0).count();
        double kms = chrono::duration<double, milli>(t2 - t1).count();
        double VE = n + 2.0 * m, ElgE = m * log2((double)m);
        cout << setw(9) << n << setw(10) << m << fixed << setprecision(3)
             << setw(12) << bms << setw(12) << kms << setprecision(4)
             << setw(15) << 1000.0 * bms / VE
             << setw(18) << 1000.0 * kms / ElgE << "\n";
    }
    return 0;
}
```

The output:

```
=== 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
```

Now read it, because numbers that are printed but not read are not an evaluation.

Test 1 is a cross-check, not a proof. Prim and Kruskal are different algorithms that must agree on the *weight* of the minimum spanning tree, though not necessarily on its edges (§11.12.1). Two thousand agreements is strong evidence that neither implementation has a systematic bug. It is not evidence that both are correct: two wrong implementations could agree. Say so — honesty about the strength of your evidence is part of the evidence.

Test 2 is a postcondition, and it is the better kind of evidence, because it checks a property that must hold of the *true* answer rather than of another implementation. If d is the breadth-first distance, then for every edge $\{u, v\}$

$$|\, d[u] - d[v] \,| \leq 1. \tag{12.4}$$

Equation (12.4) holds because an edge cannot skip a level: if $d[v] \geq d[u] + 2$, then when u was dequeued it would have discovered v at distance $d[u] + 1$, a contradiction. Zero violations across two thousand graphs is a real result.

Test 3 is the scaling claim, tested. The design predicted $\Theta(V + E)$ for BFS and $\Theta(E\lg E)$ for Kruskal. The last two columns divide the measured time by exactly those predictions. If the prediction is right, those columns should stay roughly constant as the input grows sixteen-fold. Absolute times depend on the machine and vary by a few tens of percent from run to run, so it is the two ratio columns, not the milliseconds, that carry the claim. Kruskal's column reads 0.0043,0.0041,0.0037,0.0037,0.0033: essentially flat, drifting slightly *downward* as a fixed start-up cost is spread over more work. BFS reads 0.0047,0.0044,0.0049,0.0052,0.0065: flat for the first three sizes and then rising, by about 38% between the smallest input and the largest.

That rise is not a bug and it is not a failure of the analysis, and explaining it correctly is the mark of an evaluation that has understood its own numbers.

> **WHY THE LINEAR ALGORITHM STOPS LOOKING LINEAR.** $\Theta(V + E)$ counts *operations*. It does not count cache misses. At $|V| = 32\,000$ and $|E| = 128\,000$ the adjacency structure no longer fits in the processor's fastest cache, so each of the same number of memory accesses costs more nanoseconds than it did before. The operation count is still linear; the *time per operation* has gone up. An honest evaluation reports both, and predicts that the ratio would settle again once the whole structure is out of cache.

The business answer completes the evaluation: the backbone is 6.9 km against 12.3 km of candidate roads, a 43.9% reduction in daily distance driven, and six of the fourteen zones sit within the two-hop promise while two cannot be served at all. Those three numbers, not the timings, are what the laundry's owner reads.

12.2.11 Stage 9: the conclusion

One paragraph, four sentences. It says what was built, what the numbers were, what the limits are, and what comes next.

> **CONCLUSION.** Rute Laundry models the Profitina Laundry pickup problem as three graph problems on one data set, and solves them with breadth-first search, Kruskal's algorithm over a disjoint-set forest, and Kahn's topological sort, at a total predicted cost of $\Theta(m\lg m)$ dominated by the edge sort. On the fourteen-zone instance it places six zones inside the two-hop promise, identifies two zones that no courier route reaches at all, selects an eleven-road backbone of 6.9 km against 12.3 km of candidates — a 43.9% reduction — and shows a 121-minute critical path through the wash pipeline with 106 minutes of slack on the invoicing

stage. Its main limitation is the unweighted hop model: a ride through Gebang at 07:30 is not the same cost as the same ride at 14:00, so the two-hop promise is a lower bound on travel time and not a guarantee. The natural next step is therefore Chapter 13: replace the hop count with a travel-time weight and BFS with Dijkstra's algorithm, and re-run the same evaluation harness against the new answer.

Notice what that paragraph does *not* do. It does not summarize the algorithms — the earlier stages already did. It does not apologize. It names a real limitation and a specific next step, and that is the entire content of a conclusion.

12.2.12 Five pitfalls of a design project, and a check yourself

1. **The decorative algorithm.** Applying BFS to a board so small that any method works. Apply the load-bearing test of §12.2.2 *before* you start building.
2. **The design written last.** It reads as description rather than argument, and it contains no rejected alternative. Write stages 1 to 6 first, and keep the version you wrote first.
3. **The missing evaluation.** A program run only on its own example input has not been shown correct. Two hours — a brute-force oracle or a postcondition, a few thousand random inputs, timings at five sizes — turns an assertion into evidence.
4. **The unread measurement.** A timing table printed without the ratio to the prediction, or with the ratio but no comment on its drift, is data, not evaluation.
5. **The unchecked degenerate input.** What does the program do on an empty graph, a single vertex, a disconnected graph, a cyclic "DAG"? **Listing 12.1** answers the last two: it prints two components rather than crashing, and it prints `emitted 10 of 10` rather than assuming acyclicity. Each of those is one line of code and one sentence of the write-up.

CHECK YOURSELF. (a) The operations team re-surfaces the Sukolilo–Menur road, and its length drops from 8 to 5 hundred meters. Rerun Kruskal on **Table 12.4**: which road leaves the backbone, which joins it, and what is the new total? (b) Separately, invoicing is moved to a slower system and now takes 110 minutes instead of 4. What happens to the turnaround time and to the critical path?

Answer. (a) Sukolilo–Menur (5) is now examined third, tied with Keputih–Gebang and Mulyosari–Kalijudan, and is accepted. Menur–Gebang (7) is then rejected, because Menur and Gebang are already joined through Sukolilo and Keputih. The backbone still has eleven roads and weighs $69 - 7 + 5 = 67$ hundred meters, 6.7 km. (b) By Equation (12.2), invoice now finishes at $5 + 110 = 115$, exactly when fold does, so pack finishes at $\max(115,115) + 6 = 121$: the turnaround time is **unchanged**, but there are now **two** critical paths, the old one and intake → weigh → invoice → pack. The 106 minutes of slack have been used up completely; one more minute of invoicing would delay every bag.

12.3 Further Worked Problems I: Greedy Algorithms

12.3.1 How the worked problems are laid out

Sections 12.3 to 12.6 work twenty-three problems, lettered A to W and grouped by the chapter they draw on. **Table 12.9** lists them all, so that a reader revising one weak area can go straight to it.

Table 12.9 The twenty-three worked problems of Sections 12.3 to 12.6, the section of this chapter where each is worked, and the tool from Chapters 9 to 11 that solves it.

Problem	Topic	Section	Main tool, and where it was taught
A	Making change: a greedy rule disproved, another proved	12.3.2	greedy failure (§9.3); exchange argument (§9.4.7)
B	Activity selection, run and proved	12.3.3	earliest finish time; greedy-choice property (§9.4)
C	Minimizing maximum lateness	12.3.4	exchange of adjacent inversions
D	A Huffman code, built step by step	12.3.5	Huffman's algorithm; $B(T)$ two ways (§9.7)
E	The two knapsacks	12.3.6	fractional greedy vs 0-1 dynamic programming (§9.6)
F	A matroid at work: unit-time tasks with deadlines	12.3.7	matroid greedy algorithm (§9.8)
G	Breadth-first search, traced	12.4.2	BFS, d and π (§10.7)
H	Depth-first search of a digraph, every arc classified	12.4.3	timestamps, edge classes, parenthesis theorem (§10.8)
I	Depth-first search of an undirected graph	12.4.4	discovery and finish table; tree and back edges only (§10.8.10)
J	Connected components	12.4.5	repeated BFS; circuit rank (§10.9.1)
K	Bipartiteness, and its certificate	12.4.6	two-coloring; odd cycles (§10.9.3)
L	Cycle detection, undirected and directed	12.4.7	the parent rule; the gray rule (§10.9.2, §11.2)
M	Is it a DAG?	12.5.2	the DAG theorem (§11.2.3)
N	Topological sort by finishing time	12.5.3	decreasing f (§11.3.4)
O	Topological sort by Kahn's algorithm	12.5.4	in-degrees and a queue (§11.3.9)
P	Counting the topological orders	12.5.5	recursion on sets; Hamiltonian paths
Q	The critical path	12.5.6	relaxation in topological order
R	Kruskal, with a disjoint-set trace	12.6.2	`Find`, `Union`, ranks (§11.10)
S	Prim, with the key array	12.6.3	key and π (§11.9)
T	Prim, with a binary heap	12.6.4	lazy `DecreaseKey` (§11.9.10)
U	The cut property and the cycle property	12.6.5	exchange arguments on trees (§11.7)
V	Union by rank and path compression	12.6.6	the rank bound (§11.10.5–7)
W	Is the minimum spanning tree unique?	12.6.7	deletion test; cycle property (§11.12)

Every problem has the same shape, so that you always know where you are:

- **Problem** — the statement, with every number you need.
- **What it tests** — the tool, and where in the book it was introduced.
- **Solution** — step by step, with every intermediate value written down, and the trap that catches people named where it occurs.
- **Answer** — the result, stated plainly.
- **Program** — each group of problems ends with a complete C++17 program that computes every answer in the group, quoted with the output it really printed; problems A, F, I and T have programs of their own.
- **Check yourself** — a variant for you to do alone, with its answer.

This section works six problems on greedy algorithms. The first shows greed failing, the next four show it succeeding and prove why, and the sixth shows the general theory — the matroid — that explains a whole family of successes at once.

12.3.2 Worked Problem A — making change: a greedy rule disproved, another proved

Problem. A laundry kiosk hands out change in tokens of value 1, 7 and 10. The obvious rule, **Pseudocode 12.2**, is *always hand over the largest token that still fits*.

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
 count = 0
 for i = 1 to k
     while x >= c_i
         x = x - c_i
         count = count + 1
 return count
```

(a) Run the rule on $x = 14$. (b) Find the true minimum number of tokens for $x = 14$. (c) For which amounts $1 \le x \le 30$ does the rule fail, and what is the smallest? (d) Explain, using the two ingredients of §9.5, which ingredient is missing. (e) Prove that the same rule *is* optimal for the tokens {1,5,10}.

What it tests. That a greedy rule is a *claim* and needs a proof (§9.2.2); that one counterexample kills it (§9.3.3); and that an exchange argument can prove it when it is true (§9.4.7).

Solution.

(a) The greedy run on 14. The largest token not exceeding 14 is 10: hand it over, 4 remains. 7 does not fit into 4. Then 1, four times: 3, 2, 1, 0 remain. Total: $10 + 1 + 1 + 1 + 1$, **five tokens**.

(b) The true minimum for 14. Two tokens of 7 make 14 exactly. One token cannot make 14, since no token is worth 14. So the minimum is **two tokens**, and greed used two and a half times as many.

(c) Every failure up to 30. To know the true minimum for every amount we need more than cleverness; we need a method that is certainly right. Let $M(x)$ be the fewest tokens that pay exactly x. The last token handed over is some $c \le x$, and what remains, $x - c$, must itself be paid optimally, so

$$M(0) = 0, \qquad M(x) = 1 + \min_{\substack{c \in \text{coins} \\ c \le x}} M(x - c) \quad (x \ge 1). \tag{12.5}$$

Equation (12.5) is the optimal-substructure recurrence of dynamic programming, and computing it for $x = 1,2, \ldots ,30$ in order costs $30 \times 3 = 90$ evaluations. Working the first few by hand: $M(1) = 1$, $M(2) = 2$, ..., $M(6) = 6$ (only 1-tokens fit); $M(7) = 1 + \min(M(6), M(0)) = 1 + 0 = 1$; $M(10) = 1 + \min(M(9), M(3), M(0)) = 1$; $\quad M(14) = 1 + \min(M(13), M(7), M(4)) = 1 + \min(4,1,4) = 2$. **Listing 12.3** does the rest and prints greed's answer beside the optimum.

Listing 12.3 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`.

```
// 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 12.3 (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;
}
```

```
coins {1, 7, 10}, amounts 1..30
  x  greedy              #g  optimum            #o
  1  1                    1  1                    1
  2  1+1                  2  1+1                  2
  3  1+1+1                3  1+1+1                3
  4  1+1+1+1              4  1+1+1+1              4
  5  1+1+1+1+1            5  1+1+1+1+1            5
  6  1+1+1+1+1+1          6  1+1+1+1+1+1          6
  7  7                    1  7                    1
  8  7+1                  2  1+7                  2
  9  7+1+1                3  1+1+7                3
 10  10                   1  10                   1
 11  10+1                 2  1+10                 2
 12  10+1+1               3  1+1+10               3
 13  10+1+1+1             4  1+1+1+10             4
 14  10+1+1+1+1           5  7+7                  2   <- greed loses
 15  10+1+1+1+1+1         6  1+7+7                3   <- greed loses
 16  10+1+1+1+1+1+1       7  1+1+7+7              4   <- greed loses
 17  10+7                 2  7+10                 2
 18  10+7+1               3  1+7+10               3
 19  10+7+1+1             4  1+1+7+10             4
 20  10+10                2  10+10                2
 21  10+10+1              3  1+10+10              3
 22  10+10+1+1            4  1+1+10+10            4
 23  10+10+1+1+1          5  1+1+1+10+10          5
 24  10+10+1+1+1+1        6  7+7+10               3   <- greed loses
 25  10+10+1+1+1+1+1      7  1+7+7+10             4   <- greed loses
 26  10+10+1+1+1+1+1+1    8  1+1+7+7+10           5   <- greed loses
 27  10+10+7              3  7+10+10              3
 28  10+10+7+1            4  1+7+10+10            4
 29  10+10+7+1+1          5  1+1+7+10+10          5
 30  10+10+10             3  10+10+10             3
  amounts on which greed loses: 6 of 30; the smallest is 14

coins {1, 5, 8}, amounts 1..100
  amounts on which greed loses: 47 of 100; the smallest is 10
coins {1, 5, 10, 25}, amounts 1..100
  amounts on which greed loses: 0 of 100; greed is optimal on every amount tested
```

Greed loses on exactly six amounts up to 30 — 14, 15, 16, 24, 25 and 26 — and the smallest is **14**. The pattern is visible in the table: whenever the amount is 4, 5 or 6 more than a multiple of 10, two 7s beat one 10 plus a pile of 1s.

(d) The missing ingredient. Optimal substructure is present — Equation (12.5) *is* an optimal-substructure statement, and it is why dynamic programming gets the right answer. What is missing is the **greedy-choice property**: it is *not* true that some optimal payment of 14 contains the largest token that fits, because the only optimal payment, $7 + 7$, contains no 10 at all. Greed commits to the 10 and can never take it back.

(e) Proof that greed is optimal for $\{1,5,10\}$. Take any optimal payment of x, using a tens, b fives and e ones, so $x = 10a + 5b + e$ with $a + b + e$ as small as possible.

- $e \leq 4$: otherwise replace five ones by one five, saving four tokens — contradicting optimality.
- $b \leq 1$: otherwise replace two fives by one ten, saving one token.
- Hence the fives and ones together are worth $5b + e \leq 5 + 4 = 9 < 10$.

So $10a \leq x < 10a + 10$, which forces $a = \lfloor x/10 \rfloor$ — exactly the number of tens greed takes. The remainder $x - 10a < 10$ is then paid with $b = \lfloor (x - 10a)/5 \rfloor$ fives (since $e \leq 4$ forces it) and the rest in ones, which is again exactly what greed does. Every optimal payment *is* the greedy payment. ■

The same argument breaks for {1,7,10} at the second bullet: two 7s are worth 14, and no single token is worth 14, so there is no exchange that shortens a payment containing two 7s — and that is precisely where the counterexample lives.

How far must one search before declaring a coin system safe for greed? Not far. Kozen and Zaks proved that if greed fails at all, it already fails on some amount smaller than the sum of the two largest coins [5]; for {1,7,10} that bound is $10 + 7 = 17$, and indeed the first failure is 14. Pearson later gave a polynomial-time test that decides the question for any coin system [6]. A system on which greed is optimal for every amount is called **canonical**.

Answer. Greed pays 14 with five tokens where two suffice; it fails on 14, 15, 16, 24, 25, 26 in the range 1 to 30. The greedy-choice property is what fails. For {1,5,10} an exchange argument proves greed optimal, and the audit of the US system {1,5,10,25} finds no failure up to 100.

> **CHECK YOURSELF.** For the tokens {1,5,8}, find the smallest amount on which greed fails, and give both payments.
>
> *Answer.* $x = 10$. Greed pays $8 + 1 + 1$ (three tokens); the optimum is $5 + 5$ (two). The second audit of **Listing 12.3** confirms 10 as the smallest failure, and finds that greed loses on 47 of the amounts from 1 to 100.

12.3.3 Worked Problem B — activity selection, run and proved

Problem. Ten jobs compete for a single industrial dryer. Job a_i occupies the dryer over the half-open interval $[s_i, f_i)$:

Table 12.10 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

(a) Verify that the jobs in **Table 12.10** are listed in non-decreasing order of finish time, and give the duration of each. (b) Run the earliest-finish-time greedy rule by hand, showing every comparison, and state the set selected. (c) Run the earliest-start-time rule by hand and state its answer. (d) State and prove the greedy-choice property for this problem by an exchange argument.

What it tests. Whether you can run a greedy rule mechanically *and* prove it right (§9.4.7). Part (b) is bookwork; part (d) is the only place in the problem where a proof is demanded, and it is the part that matters.

Solution.

(a) The check, and the durations. The finish times read 3,4,6,7,9,11,12,13,14,16. Each is strictly greater than the one before, so the list is sorted — a stronger condition than the algorithm needs, which is only non-decreasing. Subtracting gives the durations, which **Table 12.11** records together with the number of other jobs each one clashes with (needed when the fewest-conflicts rule of §9.4.3 is tried in the program):

Table 12.11 The ten dryer jobs with their durations and the number of other jobs each one clashes with. Jobs a_1, a_2, a_4 and a_{10} tie for the fewest conflicts at two.

i	1	2	3	4	5	6	7	8	9	10
$f_i - s_i$	3	2	5	2	5	3	5	3	5	4
conflicts	2	2	4	2	4	4	4	4	4	2

(b) Earliest finish time, by hand. **Pseudocode 12.3** is the iterative rule of §9.4.11. It keeps one variable, F, the time the dryer next becomes free, which starts at 0, and it walks the list in index order — which is finish-time order, which is why sorting first mattered — taking a job whenever $s_i \geq F$.

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
A = {a_1}
F = f_1
for i = 2 to n
    if s_i >= F
        A = A + {a_i}
        F = f_i
return A
```

- $i = 1$: $s_1 = 0 \geq 0 = F$. **Take** a_1. $F \leftarrow f_1 = 3$. Selected: $\{a_1\}$.
- $i = 2$: $s_2 = 2 \geq 3$? No. Skip.
- $i = 3$: $s_3 = 1 \geq 3$? No. Skip.
- $i = 4$: $s_4 = 5 \geq 3$? Yes. **Take** a_4. $F \leftarrow 7$. Selected: $\{a_1, a_4\}$.
- $i = 5$: $s_5 = 4 \geq 7$? No. Skip.
- $i = 6$: $s_6 = 8 \geq 7$? Yes. **Take** a_6. $F \leftarrow 11$. Selected: $\{a_1, a_4, a_6\}$.
- $i = 7$: $s_7 = 7 \geq 11$? No. $i = 8$: $10 \geq 11$? No. $i = 9$: $9 \geq 11$? No.
- $i = 10$: $s_{10} = 12 \geq 11$? Yes. **Take** a_{10}. $F \leftarrow 16$.

Answer: $\{a_1, a_4, a_6, a_{10}\}$, of size **4**, occupying the dryer over [0,3), [5,7), [8,11) and [12,16). Ten comparisons, one per job — the sweep is $\Theta(n)$ once the list is sorted.

(c) Earliest start time, by hand. Sort by start time: $a_1(0), a_3(1), a_2(2), a_5(4), a_4(5), a_7(7), a_6(8), a_9(9), a_8(10), a_{10}(12)$.

- Take $a_1 = [0,3)$. Discard everything overlapping it: $a_3 = [1,6)$ and $a_2 = [2,4)$ both do.
- Of what remains the earliest-starting is $a_5 = [4,9)$. **Take it.** Discard $a_4 = [5,7)$, $a_7 = [7,12)$ and $a_6 = [8,11)$, all of which overlap [4,9).
- Of what remains the earliest-starting is $a_9 = [9,14)$. **Take it.** Discard $a_8 = [10,13)$ and $a_{10} = [12,16)$.
- Nothing remains.

Answer: $\{a_1, a_5, a_9\}$, of size **3**. The rule is **wrong**, and this instance proves it, because part (b) produced four. The one long job a_5, chosen because it happened to start early, destroyed three later options.

(d) The greedy-choice property, proved. Write $S_k = \{a_i \in S : s_i \geq f_k\}$ for the set of jobs that can still run after a_k has finished.

> **THEOREM (greedy choice).** Let S_k be non-empty and let a_m be the job in S_k with the earliest finish time. Then a_m belongs to *some* maximum-size set of mutually compatible jobs drawn from S_k.

Proof. Let A_k be any maximum-size compatible subset of S_k, and let a_j be the job in A_k that finishes earliest. Two cases.

Case 1: $a_j = a_m$. Then $a_m \in A_k$ and there is nothing to prove.

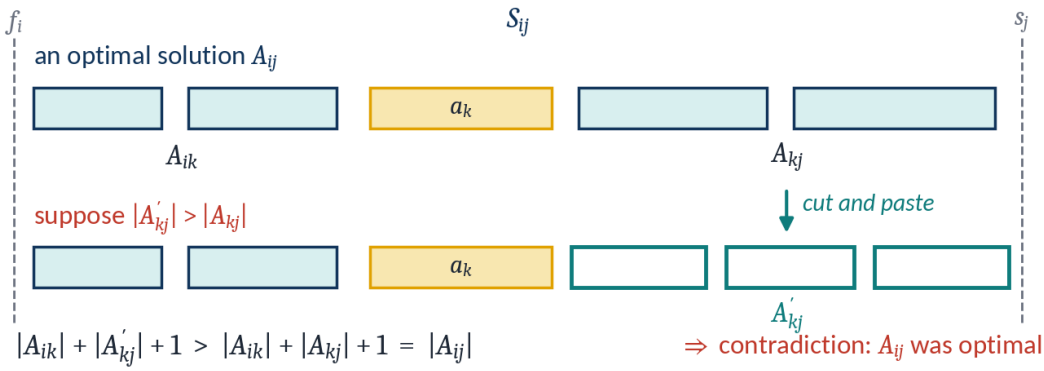
Case 2: $a_j \neq a_m$. Form

$$A'_k = (A_k \setminus \{a_j\}) \cup \{a_m\}. \tag{12.6}$$

We must check that the set A'_k of Equation (12.6) is still compatible. The jobs of $A_k \setminus \{a_j\}$ are mutually compatible because A_k was. Every one of them starts at or after f_j, because a_j was the earliest to finish in A_k and the jobs are disjoint. And a_m was chosen to finish earliest in all of S_k, so $f_m \leq f_j$. Combining, every job in $A_k \setminus \{a_j\}$ starts at or after $f_j \geq f_m$, so none of them overlaps a_m. Hence A'_k is compatible.

Finally $|A'_k| = |A_k|$, since we removed one job and added one. So A'_k is also a maximum-size compatible subset of S_k, and it contains a_m. ■

The picture of this substitution is **Figure 12.5**: if a strictly better set existed for the remaining subproblem, substituting it back would contradict the optimality we started from.



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 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.

THE TRAP IN PART (d). The theorem says *some* optimal solution contains the greedy choice, not *every* one. On this very instance $\{a_2, a_4, a_6, a_{10}\}$ is also optimal and does not contain a_1. A proof that begins "let A be the optimal solution" has already gone wrong, because it assumes the optimum is unique. Write "let A_k be *a* maximum-size compatible subset".

An alternative method. The same answer follows from dynamic programming. Define $c[i,j]$ as the size of the largest compatible set drawn from S_{ij}, the jobs that fit strictly between a_i and a_j; then $c[i,j] = \max_{a_k \in S_{ij}}\{c[i,k] + c[k,j] + 1\}$, which is $\Theta(n^3)$ to fill. The greedy theorem says that the maximizing k is always the earliest-finishing one, so the max collapses to a single candidate and the table collapses to a sweep. Knowing the recurrence and knowing why it is unnecessary is worth more than either alone.

Answer. Earliest finish time selects $\{a_1, a_4, a_6, a_{10}\}$, of size 4, which is optimal; earliest start time selects only 3. The exchange of Equation (12.6) proves that the earliest-finishing job always belongs to some optimal set.

The check-yourself variant below also uses the eleven activities of §9.4.2, shown again as **Figure 12.6**.

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 $f_1 \le f_2 \le \cdots \le f_{11}$.

CHECK YOURSELF. (a) Run the *latest start time first* rule on **Table 12.10**: repeatedly take the remaining job that starts last and discard everything overlapping it. Is the answer optimal, and why? (b) Run earliest finish time on the eleven activities of **Figure 12.6**.

Answer. (a) By start time, descending: $a_{10}(12)$, $a_8(10)$, $a_9(9)$, $a_6(8)$, $a_7(7)$, $a_4(5)$, $a_5(4)$, $a_2(2)$, $a_3(1)$, $a_1(0)$. Take $a_{10} = [12,16)$; a_8 and a_9 overlap it. Take $a_6 = [8,11)$; a_7 overlaps it. Take $a_4 = [5,7)$; a_5 overlaps it. Take $a_2 = [2,4)$; a_3 and a_1 overlap it. Result $\{a_2, a_4, a_6, a_{10}\}$, size 4 — **optimal**, and it is the *other* optimum named in the warning above. It must be optimal: reverse time (map every t to $-t$), and latest-start-first becomes earliest-finish-first on the mirrored instance, which the theorem covers. (b) $\{a_1, a_4, a_8, a_{11}\}$, of size 4; $\{a_2, a_4, a_9, a_{11}\}$ is equally optimal.

12.3.4 Worked Problem C — minimizing maximum lateness

Problem. Six jobs must run one at a time on a single machine, all available at time 0. Job J_i needs p_i units of processing and is due at d_i:

Table 12.12 The six jobs of Worked Problem C, with processing times and deadlines.

Job	J_1	J_2	J_3	J_4	J_5	J_6
p_i	3	2	1	4	3	2
d_i	6	8	9	9	14	15

If a job completes at time C_i, its **lateness** is $L_i = C_i - d_i$ (negative when it is early), and the objective is

$$L_{\max} = \max_{1 \le i \le n} (C_i - d_i) \;\rightarrow\; \min. \tag{12.7}$$

(a) Schedule the jobs of **Table 12.12** by *earliest deadline first* and compute every C_i, every L_i and $L_{\max}$. (b) Do the same for *shortest processing time first*, and compare. (c) Prove that earliest deadline first minimizes Equation (12.7), by an exchange argument on adjacent inversions.

What it tests. Whether you can build an exchange argument yourself on a problem you have not seen proved. The activity-selection proof of §9.4.7 is the model; this one is its sibling, and part (c) is the heart of it. The problem and its proof are treated in Kleinberg and Tardos [7].

Solution.

(a) Earliest deadline first. Sort by d_i: $J_1(6), J_2(8), J_3(9), J_4(9), J_5(14), J_6(15)$. The machine never idles, so C_i is the running sum of the processing times; **Table 12.13** works it through.

Table 12.13 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

$$L_{\max} = \max\{-3, -3, -3, +1, -1, 0\} = \mathbf{1}.$$

(b) Shortest processing time first. Sort by p_i: $J_3(1), J_2(2), J_6(2), J_1(3), J_5(3), J_4(4)$, and repeat the arithmetic as **Table 12.14**.

Table 12.14 Shortest processing time first on the same six jobs. $L_{\max} = 6$, six times worse than earliest deadline first, even though the *total* completion time is smaller.

Order	Job	p_i	C_i	d_i	L_i
1	J_3	1	1	9	-8
2	J_2	2	3	8	-5
3	J_6	2	5	15	-10
4	J_1	3	8	6	$+2$
5	J_5	3	11	14	-3
6	J_4	4	15	9	$+6$

$L_{\max} = 6$. The comparison is the point: sorting by the wrong key is not a small mistake. Shortest-processing-time-first is in fact optimal for a *different* objective — minimizing the average completion time — and the two objectives disagree here by a factor of six.

(c) The proof. Say a schedule has an **inversion** if some job i is scheduled before some job j although $d_i > d_j$.

Step 1: there is an optimal schedule with no idle time. Removing an idle gap moves every later job earlier, which can only decrease completion times and therefore lateness.

Step 2: swapping an adjacent inverted pair does not increase $L_{\max}$. Suppose job i runs immediately before job j, with $d_i > d_j$, and everything before i completes at time t. Before the swap,

$$C_i = t + p_i, \qquad C_j = t + p_i + p_j.$$

After the swap,

$$C'_j = t + p_j, \qquad C'_i = t + p_i + p_j.$$

No other job moves, so only L_i and L_j can change.

For j: $L'_j = t + p_j - d_j \le t + p_i + p_j - d_j = L_j$, because $p_i \ge 0$. So j is no later than before.

For i: $L'_i = t + p_i + p_j - d_i$. Compare it with the *old* lateness of j, $L_j = t + p_i + p_j - d_j$. The two expressions differ only in the deadline subtracted, and $d_i > d_j$, so

$$L'_i = t + p_i + p_j - d_i \;<\; t + p_i + p_j - d_j = L_j. \tag{12.8}$$

By Equation (12.8) and the line before it, $\max(L'_i, L'_j) \le \max(L_i, L_j)$, and since nothing else changed, $L_{\max}$ did not increase.

Step 3: any schedule can be turned into the earliest-deadline schedule by adjacent swaps of inverted pairs. This is exactly bubble sort: a schedule with no inversion is sorted by deadline, and while an inversion exists there is an *adjacent* inverted pair (if i precedes j with $d_i > d_j$ but every adjacent pair between them were in order, the deadlines would be non-decreasing from i to j, contradicting $d_i > d_j$). At most $\binom{6}{2} = 15$ swaps suffice here, $\binom{n}{2}$ in general.

Putting the three steps together: start from any optimal schedule and bubble it into deadline order; by Step 2 no swap increased $L_{\max}$, so the earliest-deadline schedule is at least as good. Hence it is optimal. ■

The trap. Schedules that differ only in the order of jobs with *equal* deadlines have the same $L_{\max}$, so "the" optimum is not unique. Here J_3 and J_4 both have $d = 9$, and swapping them gives $L_{\max} = 1$ again, by a different job. Brute force over all $6! = 720$ orders, in the program of §12.3.8, confirms both the value and the multiplicity: the optimum is 1 and **20** of the 720 orders attain it.

Answer. Earliest deadline first gives $L_{\max} = 1$, which is optimal; shortest processing time first gives 6.

> **CHECK YOURSELF.** Change d_4 from 9 to 7 and redo part (a). Which jobs are now late, and is $L_{\max}$ still optimal?
>
> *Answer.* The order becomes $J_1(6), J_4(7), J_2(8), J_3(9), J_5(14), J_6(15)$ with $C = 3,7,9,10,13,15$ and $L = -3,0,+1,+1,-1,0$. Now J_2 and J_3 are each late by one, and J_4 is exactly on time. $L_{\max} = 1$, and by part (c) it is optimal; a brute force over the 720 orders finds that 4 of them attain it.

12.3.5 Worked Problem D — a Huffman code, built step by step

Problem. A machine log of 100 000 events uses seven symbols, with these frequencies in thousands: R 38, W 22, D 15, F 12, P 8, X 4, Q 1. (a) How many bits does a fixed-length code need, and what does the whole log cost? (b) Run Huffman's algorithm, showing the queue before every merge and the label of every node created, and draw the tree. (c) Give the codeword and its length for each symbol, and compute the cost $B(T)$ twice — once by summing over the leaves and once by summing the internal-node labels. (d) State the saving as a percentage, and explain why no prefix-free code can do better on this data.

What it tests. Mechanical accuracy, and the two independent ways to compute $B(T)$ (§9.7.5). Part (c) asks for both on purpose: if they disagree, there is an arithmetic slip, and you can find it yourself.

Solution.

(a) Fixed length. Seven symbols need $\lceil \lg 7 \rceil$ bits. Since $\lg 4 = 2$ and $\lg 8 = 3$, we have $2 < \lg 7 < 3$, so the ceiling is 3. Check that the frequencies sum correctly first: $38 + 22 + 15 + 12 + 8 + 4 + 1 = 100$ thousand, so the log holds 100 000 events, as stated. At three bits each,

$$3 \times 100\,000 = 300\,000 \text{ bits} = 300 \text{ kbit}.$$

(b) Huffman, merge by merge. **Pseudocode 12.4** is the procedure of §9.7.6. The queue Q is a min-priority queue on frequency; each merge removes the two smallest trees and inserts a new internal node whose label is their sum.

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
```

Initially the queue holds seven leaves. **Table 12.15** records every merge.

Table 12.15 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

Follow the merges and the tree assembles itself. Merge 1 makes Q and X siblings under a node labeled 5. Merge 2 hangs that node beside P under 13. Merge 3 hangs 13 beside F under 25. Merge 4 makes D and W siblings under 37 — a separate little tree, because at that moment 15 and 22 were the two smallest items in the queue. Merge 5 joins the two subtrees 25 and 37 under 62, and merge 6 puts R beside 62 under the root, 100. Drawn with the first-extracted child on the left (edge label 0) and the second on the right (edge label 1), the tree is:

```
      (100)
   0/      \1
 R:38      (62)
        0/       \1
      (25)        (37)
   0/    \1     0/    \1
 F:12    (13)  D:15   W:22
       0/    \1
      (5)    P:8
   0/    \1
 Q:1     X:4
```

Figure 12.7 shows the same process, panel by panel, on the six-symbol data of §9.7; the shape of the queue at each step is identical in kind to **Table 12.15**.

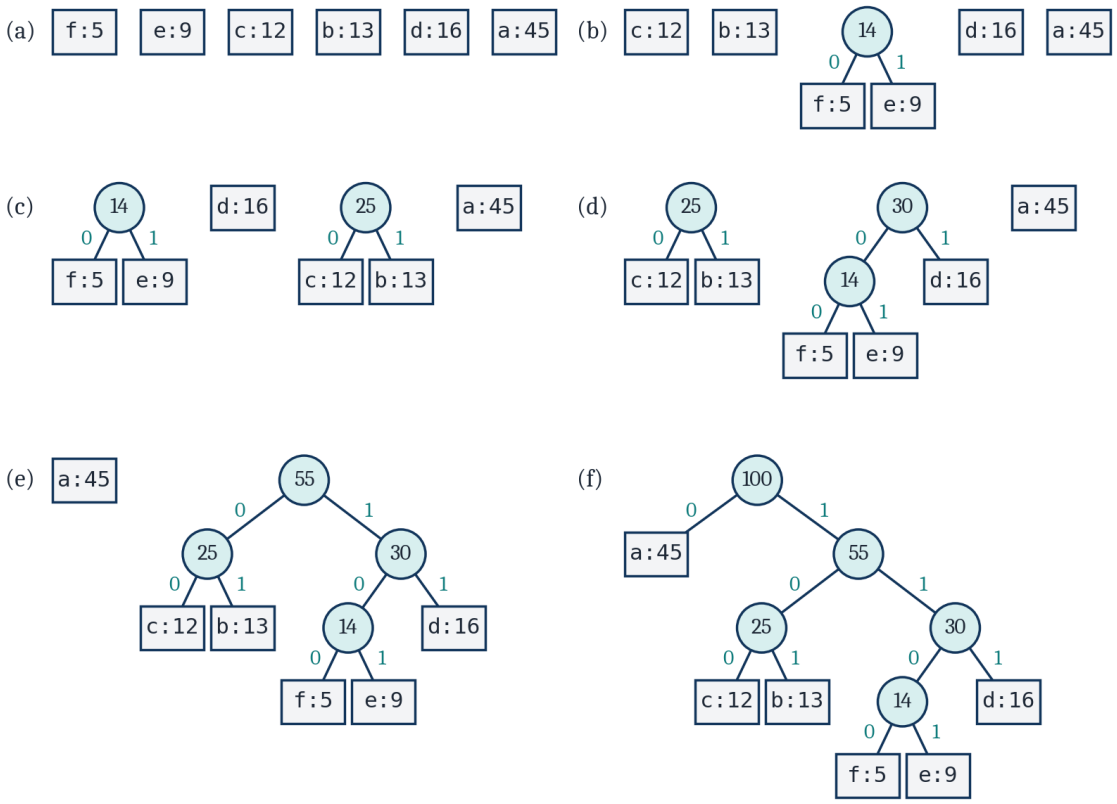


Figure 12.7 A six-panel trace of Huffman's algorithm on the six-symbol data of §9.7. Each panel shows the queue sorted into increasing frequency; at every step the two lowest-frequency trees are merged, leaves drawn as rectangles and internal nodes as circles.

(c) The codebook, and $B(T)$ computed twice. A symbol's codeword is the sequence of edge labels on the path from the root down to its leaf, so its length is the leaf's depth $d_T(c)$. Reading the tree gives **Table 12.16**.

Table 12.16 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**

Method 1, over the leaves. The cost of a tree is the number of bits it spends on the whole file,

$$B(T) = \sum_{c \in C} c\,.\,freq \cdot d_T(c), \tag{12.9}$$

and by Equation (12.9), $B(T) = 38 + 66 + 45 + 36 + 32 + 20 + 5 = 242$ kbit.

Method 2, over the internal nodes. Every internal node's label is the total frequency beneath it, and each symbol passes through exactly one internal node per bit of its codeword, so the sum of the internal labels counts every bit exactly once:

$$B(T) = \sum_{z \text{ internal}} z.freq = 5 + 13 + 25 + 37 + 62 + 100 = 242 \text{ kbit.} \tag{12.10}$$

The two methods, Equation (12.9) and Equation (12.10), agree, so the arithmetic is right.

(d) The saving, and optimality. $300 - 242 = 58$ kbit saved, which is

$$\frac{58}{300} = 0.1933\ldots = \mathbf{19.33\%}.$$

No prefix-free code can do better, by the theorem of §9.7.10 and Huffman's original argument [8]: the greedy choice is safe (there is an optimal tree in which the two least-frequent symbols are siblings at maximum depth), and the problem has optimal substructure (replacing those two by a single symbol of their combined frequency yields a smaller instance whose optimum extends to an optimum of the original). Induction on $|C|$ then gives optimality over *all* prefix-free codes.

THE TRAP. "Optimal" here means optimal *among prefix-free codes that assign a whole number of bits to each symbol, independently of context*. Arithmetic coding beats Huffman on this data by using fractional bits, and a model that predicts the next symbol from the previous one beats both. "Huffman is the best possible compression" is false; "Huffman is optimal among prefix-free per-symbol codes" is true.

Figure 12.8 compares a fixed-length code against a variable-length one on the six-symbol file of §9.7 — the same calculation as parts (a) and (c), on six symbols instead of seven.

	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.

Answer. Codewords R `0`, W `111`, D `110`, F `100`, P `1011`, X `10101`, Q `10100`; $B(T) = 242$ kbit against 300 kbit fixed-length, a saving of 19.33%, which is optimal among prefix-free codes. The program of §12.3.8 prints the same tree and encodes and decodes a 13-symbol message as a round trip.

CHECK YOURSELF. (a) Decode the bit string `10110100111110` with the codebook of **Table 12.16**. (b) Suppose Q's frequency is 6 thousand instead of 1. Rebuild the code: what are the new codeword lengths, and what is $B(T)$?

Answer. (a) Read left to right, cutting whenever a codeword is complete: `1011` P, `0` R, `100` F, `111` W, `110` D — the message is **PRFWD**. Because the code is prefix-free, there is never a choice about where to cut. (b) The total is now 105. The merges are $4 + 6 = 10$, $8 + 10 = 18$, $12 + 15 = 27$, $18 + 22 = 40$, $27 + 38 = 65$, $40 + 65 = 105$. The lengths become R 2, W 2, D 3, F 3, P 3, X 4, Q 4, and $B(T) = 38 \cdot 2 + 22 \cdot 2 + 15 \cdot 3 + 12 \cdot 3 + 8 \cdot 3 + 4 \cdot 4 + 6 \cdot 4 = 265$ kbit, which

the internal labels confirm: $10 + 18 + 27 + 40 + 65 + 105 = 265$. Notice that R has lost its one-bit codeword: it is no longer frequent *enough* relative to everything else.

12.3.6 Worked Problem E — the two knapsacks

Problem. Four items are offered; the knapsack holds 10 kg.

Table 12.17 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

(a) Compute the value density of each item in **Table 12.17** and sort. (b) Solve the *fractional* problem greedily, showing the fraction taken of each item. (c) Run the same greedy rule on the *0-1* problem, and state the value obtained. (d) Fill the dynamic-programming table $c[i, w]$ for $0 \le i \le 4$, $0 \le w \le 10$, and read off the optimal 0-1 solution. (e) By how much, in percent, does greed fall short on the 0-1 problem, and which ingredient of §9.5 does the 0-1 problem lack?

What it tests. The single most important distinction in Chapter 9 (§9.6): that a greedy rule can be provably optimal for one problem and poor for its near-twin.

Solution.

(a) Densities. $v_1/w_1 = 7/3 = 2.333$, $v_2/w_2 = 6/3 = 2.000$, $v_3/w_3 = 9/5 = 1.800$, $v_4/w_4 = 9/5 = 1.800$. Sorted: item 1, item 2, then items 3 and 4 tied.

(b) The fractional problem. Take whole items in density order while they fit, then fill the remaining space with a fraction of the next.

- Item 1: 3 kg fits in 10. Take it whole. Room left $10 - 3 = 7$. Value 7.
- Item 2: 3 kg fits in 7. Take it whole. Room left $7 - 3 = 4$. Value $7 + 6 = 13$.
- Item 3: 5 kg does **not** fit in 4. Take the fraction $4/5 = 0.8$ of it, worth $0.8 \times 9 = 7.2$. Room left 0. Value $13 + 7.2 = \mathbf{20.2}$.

That is optimal for the fractional problem: the knapsack is exactly full, and every kilogram in it is at least as dense as every kilogram left out, so no swap can improve it.

(c) The same rule on the 0-1 problem. Fractions are forbidden, so item 3 is simply skipped.

- Item 1: fits. Take. Weight 3, value 7.
- Item 2: fits. Take. Weight 6, value 13.
- Item 3: 5 kg into 4 kg of room — no. Skip.
- Item 4: 5 kg into 4 kg of room — no. Skip.

Greedy 0-1 value: **13**, using only 6 of the 10 kg. Four kilograms of capacity are wasted, and that waste is the whole failure.

(d) The dynamic-programming table. Let $c[i, w]$ be the best value obtainable from the first i items with capacity w. Either item i is left out, or it is taken and the rest of the capacity is filled optimally from the first $i - 1$ items:

$$c[i,w] = \begin{cases} 0 & i = 0 \text{ or } w = 0, \\ c[i-1,w] & w_i > w, \\ \max(c[i-1,w],\ v_i + c[i-1,\ w-w_i]) & \text{otherwise.} \end{cases} \tag{12.11}$$

Fill Equation (12.11) row by row into **Table 12.18**. Row 1 (item $v = 7, w = 3$): zero until $w = 3$, then 7 everywhere. Row 2 (item $v = 6, w = 3$): unchanged below $w = 6$; at $w = 6$, $\max(7,\ 6 + c[1,3]) = \max(7,13) = 13$. Row 3 (item $v = 9, w = 5$): at $w = 5$, $\max(7,\ 9 + c[2,0]) = 9$; at $w = 8$, $\max(13,\ 9 + c[2,3]) = \max(13,16) = 16$. Row 4 (item $v = 9, w = 5$): at $w = 10$, $\max(16,\ 9 + c[3,5]) = \max(16,18) = 18$.

Table 12.18 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**

Read the solution back by walking up the last column. $c[4,10] = 18 \neq 16 = c[3,10]$, so **item 4 is in**; drop to $c[3,10-5] = c[3,5] = 9$. Now $c[3,5] = 9 \neq 7 = c[2,5]$, so **item 3 is in**; drop to $c[2,0] = 0$ and stop. Optimal set $\{3,4\}$, weight $5 + 5 = 10$, value $9 + 9 = \mathbf{18}$. Brute force over all $2^4 = 16$ subsets confirms it.

(e) The shortfall, and the missing ingredient. Greed obtained 13 where 18 was available:

$$\frac{18-13}{18} = \frac{5}{18} = 0.2777\ldots = \mathbf{27.8\%} \text{ of the optimum lost.}$$

The ingredient the 0-1 problem lacks is the **greedy-choice property** (§9.5.2): it is *not* true that some optimal solution contains the densest item. Here the unique optimum $\{3,4\}$ excludes item 1 entirely. Optimal substructure it does have — that is exactly what Equation (12.11) exploits — which is why dynamic programming works and greed does not.

WHY THE FRACTIONAL PROBLEM IS DIFFERENT, IN ONE SENTENCE. In the fractional problem you can always fill the knapsack exactly, so "take the densest first" never wastes capacity; in the 0-1 problem an item either fits or does not, so taking a dense item can strand capacity that only a less dense item could have used. Item 1 here is dense, small, and fatal: it leaves 4 kg that nothing fits into.

Answer. Fractional optimum 20.2 by greed; 0-1 greedy 13 against the true 0-1 optimum 18 (items 3 and 4), a shortfall of 27.8%.

CHECK YOURSELF. Shrink the knapsack to 8 kg. Give the fractional optimum, the 0-1 greedy value, and the 0-1 optimum.

Answer. Fractional: items 1 and 2 whole (6 kg, value 13), then 2/5 of item 3, worth 3.6: **16.6**. 0-1 greedy: items 1 and 2, value **13**, 2 kg wasted. 0-1 optimum: **16**, by $\{1,3\}$ or equally $\{1,4\}$ (weight 8 each); read it from row 4, column 8 of **Table 12.18**. Greed now loses $3/16 = 18.75\%$.

12.3.7 Worked Problem F — a matroid at work: unit-time tasks with deadlines

Problem. On a busy morning the laundry's pressing station receives seven one-hour orders. Order a_i must be ready by the end of hour d_i; if it is late, the laundry refunds w_i thousand rupiah. Only one order can be pressed at a time.

Table 12.19 The seven pressing orders of Worked Problem F: each takes one hour, and a late order costs its refund.

Order	a_1	a_2	a_3	a_4	a_5	a_6	a_7
deadline d_i	2	4	1	3	2	4	6
refund w_i	55	45	40	35	30	20	10

(a) Explain why a set of orders can all be finished on time if and only if, for every t, at most t of them have deadline t or earlier. (b) Check the two matroid axioms on this structure, and illustrate the exchange property on a concrete pair of sets. (c) Run the matroid greedy algorithm on **Table 12.19** and give the schedule and the total refund. (d) Confirm the answer by brute force.

What it tests. The matroid of §9.8: the definition, the independence test for unit-time scheduling (§9.8.6), and the theorem that the greedy algorithm is optimal on any weighted matroid (§9.8.5), due in this form to Edmonds [9].

Solution.

(a) The independence test. Call a set A of orders **independent** if all of them can be finished on time. For a set A and an hour t, let $N_t(A)$ be the number of orders in A whose deadline is t or earlier. Then

$$A \text{ is independent} \Leftrightarrow N_t(A) \leq t \quad \text{for } t = 0,1,\dots,n. \tag{12.12}$$

Why Equation (12.12) *is necessary:* if more than t orders are due by hour t, then in the first t hours at most t of them can be pressed, so one of them is late. *Why it is sufficient:* press the orders of A in order of deadline. The order in position k of that list has at least k orders (itself and those before it) due by its own deadline d, so $k \leq N_d(A) \leq d$: it finishes at hour $k \leq d$, on time.

(b) The two matroid axioms. The ground set is the seven orders, and the independent sets are those that pass Equation (12.12).

- **Hereditary.** If A is independent and $B \subseteq A$, then every count $N_t(B) \leq N_t(A) \leq t$, so B is independent. Removing orders can only make a schedule easier.
- **Exchange.** If A and B are independent and $|A| < |B|$, some $x \in B \setminus A$ keeps $A \cup \{x\}$ independent. The general proof is in §9.8.6; here is the property at work on a pair of sets. Take $A = \{a_1, a_3\}$ (deadlines 2 and 1: $N_1 = 1$, $N_2 = 2$, independent) and $B = \{a_3, a_4, a_5\}$ (deadlines 1, 3, 2: $N_1 = 1$, $N_2 = 2$, $N_3 = 3$, independent). Then $B \setminus A = \{a_4, a_5\}$. Adding a_5 gives deadlines 2, 1, 2 and $N_2 = 3 > 2$: **not** independent. Adding a_4 gives deadlines 2, 1, 3 and $N_1 = 1$, $N_2 = 2$, $N_3 = 3$: independent. So the exchange succeeds with $x = a_4$ — the property promises that *some* element of $B \setminus A$ works, not that every one does.

(c) The greedy algorithm. **Pseudocode 12.5** is the greedy algorithm of §9.8.5, specialized to this matroid: consider the orders in decreasing order of refund, and keep each one whose addition leaves the set independent. Keeping the orders with the largest refunds on time is the same as making the total refund of the late orders as small as possible.

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)
sort the orders so that w_1 >= w_2 >= ... >= w_n
A = empty set
for i = 1 to n
    if A + {a_i} satisfies N_t <= t for every t
        A = A + {a_i}
schedule A in order of deadline, then the other orders in any order
return the schedule, and the refund sum of the orders not in A
```

The orders are already in decreasing refund order in **Table 12.19**. **Table 12.20** traces the loop; each row shows the counts $N_1, \ldots, N_7$ of the *trial* set $A \cup \{a_i\}$, with a count that breaks Equation (12.12) marked.

Table 12.20 The matroid greedy algorithm on the seven orders. Bold counts break the rule $N_t \leq t$. 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	$\{a_1, a_2, a_3, a_4, a_7\}$

Look at the rejection of a_5: with it, three orders (a_1, a_3, a_5) would be due by hour 2, and two hours cannot press three orders. The rejection of a_6 is subtler: five orders would be due by hour 4. Order a_7, the cheapest of all, is kept, because its deadline of 6 leaves room.

The schedule presses the kept orders in deadline order — a_3 (hour 1, due 1), a_1 (hour 2, due 2), a_4 (hour 3, due 3), a_2 (hour 4, due 4), a_7 (hour 5, due 6) — and then the two rejected orders, which are late whatever we do: a_5 in hour 6 and a_6 in hour 7. The refund is $30 + 20 = \mathbf{50}$ thousand rupiah, out of a possible 235.

(d) Brute force. **Listing 12.4** prints the same trace and then tries all $2^7 = 128$ subsets, keeping the independent one of greatest refund weight.

Listing 12.4 The matroid greedy algorithm for unit-time tasks with deadlines and refunds, with the independence counts printed at every step and a brute-force check over all 128 subsets. Compile with `g++ -std=c++17 -O2 -Wall -o tasks tasks.cpp`.

```
// Worked Problem F: unit-time tasks with deadlines and penalties, solved by
// the matroid greedy algorithm. Seven one-hour pressing orders at the laundry;
// order i must be ready by hour d[i], or the laundry refunds w[i] thousand rupiah.
// The program prints the independence test N_t(A) <= t at every step, the
// schedule in early-first form, and checks the answer against all 2^7 subsets.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include <numeric>
using namespace std;

static const vector<int> D = { 2, 4, 1, 3, 2, 4, 6 };        // deadlines, in hours
static const vector<int> W = { 55, 45, 40, 35, 30, 20, 10 };  // refunds, thousand rupiah
static const int n = (int)D.size();

// N[t] = number of tasks in the set whose deadline is t or earlier, t = 0..n
vector<int> counts(const vector<int>& set) {
    vector<int> N(n + 1, 0);
    for (int i : set) for (int t = D[i]; t <= n; ++t) ++N[t];
    return N;
}
bool independent(const vector<int>& set) {
    vector<int> N = counts(set);
    for (int t = 0; t <= n; ++t) if (N[t] > t) return false;
    return true;
}
string name(const vector<int>& set) {
    string s = "{";
    for (size_t k = 0; k < set.size(); ++k) s += (k ? ", a" : "a") + to_string(set[k] + 1);
    return s + "}";
}

int main() {
    vector<int> order(n); iota(order.begin(), order.end(), 0);
    stable_sort(order.begin(), order.end(), [](int a, int b) { return W[a] > W[b]; });
    cout << "task   d   w\n";
    for (int i = 0; i < n; ++i) cout << "  a" << i + 1 << setw(4) << D[i] << setw(4) <<
W[i] << "\n";
```

Listing 12.4 (Continued) *The matroid greedy algorithm for unit-time tasks with deadlines and refunds, with the independence counts printed at every step and a brute-force check over all 128 subsets. Compile with g++ -std=c++17 -O2 -Wall -o tasks tasks.cpp.*

```
    vector<int> A;
    cout << "\nconsider   N_1..N_" << n << " of the trial set   verdict\n";
    for (int i : order) {
        vector<int> trial = A; trial.push_back(i); sort(trial.begin(), trial.end());
        vector<int> N = counts(trial);
        bool ok = independent(trial);
        cout << "  a" << i + 1 << "      ";
        for (int t = 1; t <= n; ++t) cout << setw(2) << N[t] << (N[t] > t ? "!" : " ");
        cout << "   " << (ok ? "keep  " : "REJECT") << "  ";
        if (ok) A = trial;
        cout << "A = " << name(A) << "\n";
    }
    int early = 0, late = 0; vector<int> L;
    for (int i = 0; i < n; ++i) {
        if (find(A.begin(), A.end(), i) != A.end()) early += W[i];
        else { late += W[i]; L.push_back(i); }
    }
    cout << "early set " << name(A) << ", weight " << early
         << "; late set " << name(L) << ", penalty " << late << "\n";

    vector<int> sched = A;                                    // early tasks by deadline,
    stable_sort(sched.begin(), sched.end(), [](int a, int b) { return D[a] < D[b]; });
    for (int i : L) sched.push_back(i);                        // then the late ones
    cout << "schedule:";
    for (int s = 0; s < n; ++s) {
        int i = sched[s];
        cout << " a" << i + 1 << (s + 1 <= D[i] ? "" : "(late)");
    }
    cout << "\n";

    int best = -1, bestMask = 0, ties = 0;                    // brute force over 2^n
subsets
    for (int mask = 0; mask < (1 << n); ++mask) {
        vector<int> set; int w = 0;
        for (int i = 0; i < n; ++i) if (mask >> i & 1) { set.push_back(i); w += W[i]; }
        if (!independent(set)) continue;
        if (w > best) { best = w; bestMask = mask; ties = 1; } else if (w == best) ++ties;
    }
    vector<int> bs; for (int i = 0; i < n; ++i) if (bestMask >> i & 1) bs.push_back(i);
    int total = accumulate(W.begin(), W.end(), 0);
    cout << "brute force over " << (1 << n) << " subsets: best early weight " << best
         << " by " << name(bs) << " (" << ties << " optimal set" << (ties > 1 ? "s" : "")
         << "), minimum penalty " << total - best << "\n";
    return 0;
}
```

```
task   d   w
  a1   2  55
  a2   4  45
  a3   1  40
  a4   3  35
  a5   2  30
  a6   4  20
  a7   6  10

consider   N_1..N_7 of the trial set   verdict
  a1       0  1  1  1  1  1  1    keep    A = {a1}
  a2       0  1  1  2  2  2  2    keep    A = {a1, a2}
  a3       1  2  2  3  3  3  3    keep    A = {a1, a2, a3}
  a4       1  2  3  4  4  4  4    keep    A = {a1, a2, a3, a4}
  a5       1  3! 4! 5! 5  5  5    REJECT  A = {a1, a2, a3, a4}
  a6       1  2  3  5! 5  5  5    REJECT  A = {a1, a2, a3, a4}
  a7       1  2  3  4  4  5  5    keep    A = {a1, a2, a3, a4, a7}
early set {a1, a2, a3, a4, a7}, weight 185; late set {a5, a6}, penalty 50
schedule: a3 a1 a4 a2 a7 a5(late) a6(late)
brute force over 128 subsets: best early weight 185 by {a1, a2, a3, a4, a7} (1 optimal
set), minimum penalty 50
```

The brute force agrees: the best on-time set is $\{a_1, a_2, a_3, a_4, a_7\}$, it is the only optimal set, and the minimum refund is 50. **Figure 12.9** shows the seven-task instance of §9.8.7, on which the same algorithm keeps a set of weight 230 on time and pays a penalty of 50; the two instances are different, and it is a coincidence that both penalties come to 50.

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.

The cost. Sorting costs $\Theta(n\lg n)$. Each independence test, done as in the program by recounting, costs $O(n^2)$, so the loop is $O(n^3)$ as written; keeping the counts N_t in an array and updating only the entries $t \geq d_i$ brings each test to $O(n)$ and the whole algorithm to $O(n^2)$, which is the bound of §9.8.5; a disjoint-set structure lowers it further, to $O(n\lg n)$.

Answer. Keep $\{a_1, a_2, a_3, a_4, a_7\}$ on time, pressed in the order a_3, a_1, a_4, a_2, a_7; orders a_5 and a_6 are late; the refund is 50 thousand rupiah, which is optimal by the matroid theorem and by brute force.

CHECK YOURSELF. A customer agrees to collect a_5 later, so its deadline moves from 2 to 5. Rerun the greedy algorithm. Which orders are late now, and what is the refund?

Answer. a_1 to a_4 are kept as before. a_5 (deadline 5) is now kept: the counts become $N_1 = 1$, $N_2 = 2$, $N_3 = 3$, $N_4 = 4$, $N_5 = 5$, all within bounds. a_6 (deadline 4) is rejected, because N_4 would be 5. a_7 (deadline 6) is kept, with $N_6 = 6$. Only a_6 is late, and the refund falls to **20**.

12.3.8 Problems B to E as one program

Listing 12.5 answers Problems B to E mechanically, including the brute-force oracles that certify the greedy answers and the Huffman tree of §12.3.5.

Listing 12.5 Worked Problems B to E, computed: activity selection under four rules against brute force; earliest deadline first against all 720 orders; Huffman with its tree, $B(T)$ both ways and a round trip; and the knapsack, fractional and 0-1. Compile with `g++ -std=c++17 -O2 -Wall -o greedy greedy.cpp` and run `./greedy`.

```
// Worked Problems B to E of Chapter 12 -- greedy algorithms, computed.
//   B  activity selection: four rules, and the true optimum by brute force
//   C  minimizing maximum lateness: earliest deadline first, against all 6! orders
//   D  Huffman on the seven-symbol machine log, with a round trip
//   E  the knapsack, fractional and 0-1, greedy and dynamic programming
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <map>
#include <algorithm>
#include <numeric>
#include <functional>
#include <climits>
using namespace std;

// ============================================ B: activity selection
struct Act { int i, s, f; };
static const vector<Act> ACT = {
    {1,0,3}, {2,2,4}, {3,1,6}, {4,5,7}, {5,4,9},
    {6,8,11}, {7,7,12}, {8,10,13}, {9,9,14}, {10,12,16}
};
bool compatible(const Act& a, const Act& b) { return a.f <= b.s || b.f <= a.s; }

string show(const vector<int>& v) {
    string s = "{";
    for (size_t i = 0; i < v.size(); ++i) { if (i) s += ", "; s += "a" + to_string(v[i]); }
    return s + "}";
}

// a generic "repeatedly take the best remaining, drop what clashes" driver
vector<int> greedyBy(function<bool(const Act&, const Act&)> better) {
    vector<Act> left = ACT; vector<int> chosen;
    while (!left.empty()) {
        const Act* best = &left[0];
        for (const Act& a : left) if (better(a, *best)) best = &a;
        Act pick = *best; chosen.push_back(pick.i);
        vector<Act> keep;
        for (const Act& a : left) if (a.i != pick.i && compatible(a, pick))
keep.push_back(a);
        left = keep;
    }
    sort(chosen.begin(), chosen.end());
    return chosen;
}
```

Listing 12.5 (Continued) *Worked Problems B to E, computed: activity selection under four rules against brute force; earliest deadline first against all 720 orders; Huffman with its tree, B(T) both ways and a round trip; and the knapsack, fractional and 0-1. Compile with g++ -std=c++17 -O2 -Wall -o greedy greedy.cpp and run ./greedy.*

```
int conflictsOf(const Act& a, const vector<Act>& pool) {
    int c = 0; for (const Act& b : pool) if (b.i != a.i && !compatible(a, b)) ++c;
    return c;
}

void q1() {
    cout << "=== Problem B  activity selection ===\n";
    cout << "instance (already sorted by finish time):\n  i  ";
    for (const Act& a : ACT) cout << setw(4) << a.i;
    cout << "\n  s  ";
    for (const Act& a : ACT) cout << setw(4) << a.s;
    cout << "\n  f  ";
    for (const Act& a : ACT) cout << setw(4) << a.f;
    cout << "\n  d  ";
    for (const Act& a : ACT) cout << setw(4) << a.f - a.s;
    cout << "\n  c  ";
    for (const Act& a : ACT) cout << setw(4) << conflictsOf(a, ACT);
    cout << "\n";

    auto eft = greedyBy([](const Act& a, const Act& b){ return a.f < b.f; });
    auto est = greedyBy([](const Act& a, const Act& b){ return a.s < b.s; });
    auto sdu = greedyBy([](const Act& a, const Act& b){ return a.f - a.s < b.f - b.s; });
    vector<int> few;                                        // fewest conflicts, recomputed
each round
    {
        vector<Act> left = ACT;
        while (!left.empty()) {
            const Act* best = &left[0]; int bc = conflictsOf(left[0], left);
            for (const Act& a : left) { int c = conflictsOf(a, left);
                if (c < bc || (c == bc && a.i < best->i)) { bc = c; best = &a; } }
            Act pick = *best; few.push_back(pick.i);
            vector<Act> keep;
            for (const Act& a : left) if (a.i != pick.i && compatible(a, pick))
keep.push_back(a);
            left = keep;
        }
        sort(few.begin(), few.end());
    }
    cout << "  earliest finish time -> " << show(eft) << ", size " << eft.size() << "\n";
    cout << "  earliest start time  -> " << show(est) << ", size " << est.size() << "\n";
    cout << "  shortest duration    -> " << show(sdu) << ", size " << sdu.size() << "\n";
    cout << "  fewest conflicts     -> " << show(few) << ", size " << few.size() << "\n";
```

Listing 12.5 (Continued) *Worked Problems B to E, computed: activity selection under four rules against brute force; earliest deadline first against all 720 orders; Huffman with its tree, B(T) both ways and a round trip; and the knapsack, fractional and 0-1. Compile with g++ -std=c++17 -O2 -Wall -o greedy greedy.cpp and run ./greedy.*

```
    int n = (int)ACT.size(), bestSize = 0; vector<int> bestSet;
    for (int mask = 0; mask < (1 << n); ++mask) {          // brute force over all 1024
subsets
        vector<int> sel; bool ok = true;
        for (int i = 0; i < n && ok; ++i) if (mask >> i & 1) {
            for (int j : sel) if (!compatible(ACT[i], ACT[j-1])) { ok = false; break; }
            if (ok) sel.push_back(ACT[i].i);
        }
        if (ok && (int)sel.size() > bestSize) { bestSize = (int)sel.size(); bestSet = sel;
}
    }
    cout << "  brute force over all " << (1 << n) << " subsets -> optimum size "
         << bestSize << ", e.g. " << show(bestSet) << "\n";
}

// ============================================ C: minimizing maximum lateness
struct Job { string name; int p, d; };
static const vector<Job> JOB = {
    {"J1",3,6}, {"J2",2,8}, {"J3",1,9}, {"J4",4,9}, {"J5",3,14}, {"J6",2,15}
};
int maxLateness(const vector<int>& order, bool print = false) {
    int t = 0, L = INT_MIN;
    for (int k : order) { t += JOB[k].p; int l = t - JOB[k].d; L = max(L, l);
        if (print) cout << "    " << JOB[k].name << ": p=" << JOB[k].p
                        << " finishes at " << setw(3) << t << ", deadline " << setw(3)
                        << JOB[k].d << ", lateness " << setw(3) << l << "\n"; }
    return L;
}
void q2() {
    cout << "\n=== Problem C  minimizing maximum lateness ===\n";
    vector<int> edf(JOB.size()); iota(edf.begin(), edf.end(), 0);
    sort(edf.begin(), edf.end(), [](int a, int b){ return JOB[a].d < JOB[b].d; });
    cout << "  earliest deadline first:\n";
    int Ledf = maxLateness(edf, true);
    cout << "    max lateness = " << Ledf << "\n";
    vector<int> spt(JOB.size()); iota(spt.begin(), spt.end(), 0);
    stable_sort(spt.begin(), spt.end(), [](int a, int b){ return JOB[a].p < JOB[b].p; });
    cout << "  shortest processing time first:\n";
    int Lspt = maxLateness(spt, true);
    cout << "    max lateness = " << Lspt << "\n";
    vector<int> perm(JOB.size()); iota(perm.begin(), perm.end(), 0);
    int best = INT_MAX, orders = 0, ties = 0;
    do { int L = maxLateness(perm); ++orders;
         if (L < best) { best = L; ties = 1; } else if (L == best) ++ties;
    } while (next_permutation(perm.begin(), perm.end()));
    cout << "  brute force over all " << orders << " orders -> optimum " << best
         << ", attained by " << ties << " of them\n";
}
```

Listing 12.5 (Continued) *Worked Problems B to E, computed: activity selection under four rules against brute force; earliest deadline first against all 720 orders; Huffman with its tree, B(T) both ways and a round trip; and the knapsack, fractional and 0-1. Compile with g++ -std=c++17 -O2 -Wall -o greedy greedy.cpp and run ./greedy.*

```
// ============================================ D: Huffman
struct Node { int freq; char ch; int l = -1, r = -1; };
void q3() {
    cout << "\n=== Problem D  Huffman on the machine log ===\n";
    vector<pair<char,int>> F = {
        {'R',38}, {'W',22}, {'D',15}, {'F',12}, {'P',8}, {'X',4}, {'Q',1}
    };
    vector<Node> t;
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    for (auto [c, f] : F) { t.push_back({f, c}); Q.push({f, (int)t.size() - 1}); }
    int step = 0, internalSum = 0;
    while (Q.size() > 1) {
        auto [f1, x] = Q.top(); Q.pop();
        auto [f2, y] = Q.top(); Q.pop();
        t.push_back({f1 + f2, 0, x, y});
        internalSum += f1 + f2;
        cout << "  merge " << ++step << ": " << setw(3) << f1 << " + " << setw(3) << f2
             << " = " << setw(3) << f1 + f2 << "\n";
        Q.push({f1 + f2, (int)t.size() - 1});
    }
    int root = Q.top().second;
    map<char,string> code;
    function<void(int,string)> walk = [&](int v, string pre) {
        if (t[v].l == -1) { code[t[v].ch] = pre.empty() ? "0" : pre; return; }
        walk(t[v].l, pre + "0"); walk(t[v].r, pre + "1");
    };
    walk(root, "");
    // draw the tree: one line per node, indented by depth, edge label first
    cout << "  the tree (internal nodes in brackets, leaves as symbol:freq):\n";
    function<void(int,string,string)> draw = [&](int v, string indent, string edge) {
        cout << "    " << indent << edge;
        if (t[v].l == -1) { cout << t[v].ch << ":" << t[v].freq << "\n"; return; }
        cout << "(" << t[v].freq << ")\n";
        draw(t[v].l, indent + "   ", "0-"); draw(t[v].r, indent + "   ", "1-");
    };
    draw(root, "", "");
    int variable = 0, fixedBits = 0;
    cout << "  sym  freq  code        len  freq*len\n";
    for (auto [c, f] : F) {
        cout << "   " << c << setw(6) << f << "  " << left << setw(10) << code[c]
             << right << setw(4) << code[c].size() << setw(9) << f * (int)code[c].size() <<
"\n";
        variable += f * (int)code[c].size();
        fixedBits += f * 3;
    }
    cout << "  B(T) by summing over the leaves          = " << variable << " kbit\n";
```

Listing 12.5 (Continued) *Worked Problems B to E, computed: activity selection under four rules against brute force; earliest deadline first against all 720 orders; Huffman with its tree, B(T) both ways and a round trip; and the knapsack, fractional and 0-1. Compile with g++ -std=c++17 -O2 -Wall -o greedy greedy.cpp and run ./greedy.*

```
    cout << "  B(T) by summing the internal node labels = " << internalSum << " kbit\n";
    cout << "  fixed-length (3 bits, 7 symbols)         = " << fixedBits << " kbit\n";
    cout << "  saving = " << fixedBits - variable << " kbit = " << fixed << setprecision(2)
         << 100.0 * (fixedBits - variable) / fixedBits << "%\n";
    string msg = "RRWDRFPRWXQDR";
    string bits; for (char c : msg) bits += code[c];
    string back; int v = root;
    for (char b : bits) { v = (b == '0' ? t[v].l : t[v].r);
        if (t[v].l == -1) { back += t[v].ch; v = root; } }
    cout << "  round trip: \"" << msg << "\" -> " << bits.size() << " bits -> \"" << back
         << "\"  " << (back == msg ? "OK" : "FAILED") << "\n";
}

// ============================================ E: the knapsack
struct Item { int v, w; };
static const vector<Item> IT = { {7,3}, {6,3}, {9,5}, {9,5} };
static const int CAP = 10;
void q4() {
    cout << "\n=== Problem E  the knapsack, capacity " << CAP << " ===\n";
    int n = (int)IT.size();
    vector<int> ord(n); iota(ord.begin(), ord.end(), 0);
    stable_sort(ord.begin(), ord.end(), [](int a, int b) {
        return (double)IT[a].v / IT[a].w > (double)IT[b].v / IT[b].w; });
    cout << "  item  v   w   v/w\n";
    for (int i : ord) cout << setw(6) << i + 1 << setw(4) << IT[i].v << setw(4) << IT[i].w
                           << setw(7) << fixed << setprecision(3)
                           << (double)IT[i].v / IT[i].w << "\n";
    double room = CAP, frac = 0;
    cout << "  fractional greedy:";
    for (int i : ord) {
        if (room <= 0) break;
        double take = min((double)IT[i].w, room);
        frac += take / IT[i].w * IT[i].v; room -= take;
        cout << "  item" << i + 1 << " x" << setprecision(2) << take / IT[i].w;
    }
    cout << "  -> value " << setprecision(2) << frac << "\n";
    int cap = CAP, gv = 0; string gs;
    for (int i : ord) if (IT[i].w <= cap) { cap -= IT[i].w; gv += IT[i].v; gs += " item" +
to_string(i+1); }
    cout << "  0-1 greedy by density:" << gs << " -> value " << gv
         << ", weight used " << CAP - cap << "\n";
    vector<vector<int>> c(n + 1, vector<int>(CAP + 1, 0));
    for (int i = 1; i <= n; ++i)
        for (int w = 0; w <= CAP; ++w)
            c[i][w] = (IT[i-1].w > w) ? c[i-1][w]
                                      : max(c[i-1][w], IT[i-1].v + c[i-1][w - IT[i-1].w]);
    cout << "  the dynamic-programming table c[i][w]:\n      w =";
```

Listing 12.5 (Continued) *Worked Problems B to E, computed: activity selection under four rules against brute force; earliest deadline first against all 720 orders; Huffman with its tree, B(T) both ways and a round trip; and the knapsack, fractional and 0-1. Compile with g++ -std=c++17 -O2 -Wall -o greedy greedy.cpp and run ./greedy.*

```
    for (int w = 0; w <= CAP; ++w) cout << setw(4) << w;
    cout << "\n";
    for (int i = 0; i <= n; ++i) {
        cout << "  i = " << i << "  ";
        for (int w = 0; w <= CAP; ++w) cout << setw(4) << c[i][w];
        cout << "\n";
    }
    string sel; int w = CAP;
    for (int i = n; i >= 1; --i) if (c[i][w] != c[i-1][w]) { sel = " item" + to_string(i) +
sel; w -= IT[i-1].w; }
    cout << "  0-1 optimum by dynamic programming = " << c[n][CAP] << ", taking" << sel <<
"\n";
    int bv = 0, bm = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        int tv = 0, tw = 0;
        for (int i = 0; i < n; ++i) if (mask >> i & 1) { tv += IT[i].v; tw += IT[i].w; }
        if (tw <= CAP && tv > bv) { bv = tv; bm = mask; }
    }
    cout << "  brute force over " << (1 << n) << " subsets = " << bv << " (mask " << bm <<
")\n";
    cout << "  greedy loses " << bv - gv << " of " << bv << " = " << setprecision(1)
         << 100.0 * (bv - gv) / bv << "% of the 0-1 optimum\n";
}

int main() { q1(); q2(); q3(); q4(); return 0; }
```

The complete output, which certifies every hand computation above:

```
=== Problem B  activity selection ===
instance (already sorted by finish time):
  i     1   2   3   4   5   6   7   8   9  10
  s     0   2   1   5   4   8   7  10   9  12
  f     3   4   6   7   9  11  12  13  14  16
  d     3   2   5   2   5   3   5   3   5   4
  c     2   2   4   2   4   4   4   4   4   2
  earliest finish time -> {a1, a4, a6, a10}, size 4
  earliest start time  -> {a1, a5, a9}, size 3
  shortest duration    -> {a2, a4, a6, a10}, size 4
  fewest conflicts     -> {a1, a4, a6, a10}, size 4
  brute force over all 1024 subsets -> optimum size 4, e.g. {a1, a4, a6, a10}

=== Problem C  minimizing maximum lateness ===
  earliest deadline first:
    J1: p=3 finishes at   3, deadline   6, lateness  -3
    J2: p=2 finishes at   5, deadline   8, lateness  -3
    J3: p=1 finishes at   6, deadline   9, lateness  -3
    J4: p=4 finishes at  10, deadline   9, lateness   1
    J5: p=3 finishes at  13, deadline  14, lateness  -1
    J6: p=2 finishes at  15, deadline  15, lateness   0
    max lateness = 1
  shortest processing time first:
    J3: p=1 finishes at   1, deadline   9, lateness  -8
    J2: p=2 finishes at   3, deadline   8, lateness  -5
    J6: p=2 finishes at   5, deadline  15, lateness -10
    J1: p=3 finishes at   8, deadline   6, lateness   2
    J5: p=3 finishes at  11, deadline  14, lateness  -3
    J4: p=4 finishes at  15, deadline   9, lateness   6
    max lateness = 6
  brute force over all 720 orders -> optimum 1, attained by 20 of them

=== Problem D  Huffman on the machine log ===
  merge 1:   1 +   4 =   5
  merge 2:   5 +   8 =  13
  merge 3:  12 +  13 =  25
  merge 4:  15 +  22 =  37
  merge 5:  25 +  37 =  62
  merge 6:  38 +  62 = 100
  the tree (internal nodes in brackets, leaves as symbol:freq):
    (100)
       0-R:38
       1-(62)
          0-(25)
             0-F:12
             1-(13)
                0-(5)
                   0-Q:1
                   1-X:4
                1-P:8
          1-(37)
             0-D:15
```

```
              1-W:22
  sym  freq  code        len  freq*len
   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
  B(T) by summing over the leaves          = 242 kbit
  B(T) by summing the internal node labels = 242 kbit
  fixed-length (3 bits, 7 symbols)         = 300 kbit
  saving = 58 kbit = 19.33%
  round trip: "RRWDRFPRWXQDR" -> 34 bits -> "RRWDRFPRWXQDR"  OK

=== Problem E  the knapsack, capacity 10 ===
  item  v   w   v/w
     1   7   3  2.333
     2   6   3  2.000
     3   9   5  1.800
     4   9   5  1.800
  fractional greedy:  item1 x1.00  item2 x1.00  item3 x0.80  -> value 20.20
  0-1 greedy by density: item1 item2 -> value 13, weight used 6
  the dynamic-programming table c[i][w]:
      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
  0-1 optimum by dynamic programming = 18, taking item3 item4
  brute force over 16 subsets = 18 (mask 12)
  greedy loses 5 of 18 = 27.8% of the 0-1 optimum
```

Note what the first block shows: on *this* instance, three of the four activity rules return a set of size 4 and only earliest-start-time fails. That is exactly why part (d) of Problem B demanded a proof. Agreement on one instance is not correctness; the shortest-duration rule happens to be right here and is wrong on the three-activity instance of Exercise 12.7.

12.4 Further Worked Problems II: Searching a Graph

12.4.1 The two graphs of this section

Most problems in this section use the same two graphs, so they are stated once here. Throughout, adjacency lists are stored in **alphabetical order**, and the outer loop of any depth-first search considers vertices in **alphabetical order**. Those conventions matter: the *distances* a BFS computes do not depend on them, but the trees, the timestamps and the edge classes do, and a trace can only be checked against another trace made under the same conventions.

G_1 is **undirected**, with $V = \{A, \dots, I\}$ and twelve edges

$$E_1 = \{AB,\ AC,\ BD,\ BE,\ CE,\ CF,\ DG,\ EG,\ EH,\ FH,\ GI,\ HI\}.$$

G_2 is **directed**, with $V = \{A, \dots, G\}$ and eleven arcs

$$A \to B,\ A \to C,\ A \to D,\ B \to D,\ C \to D,\ C \to E,\ D \to A,\ E \to F,\ F \to E,\ G \to C,\ G \to F.$$

It is worth drawing both on paper before reading on. G_1 looks like a diamond-shaped lattice of five rows (A; B, C; D, E, F; G, H; I) with every edge running between neighboring rows; G_2 has two small loops, $A \to B \to D \to A$ and $E \rightleftarrows F$, which Problem H will find.

12.4.2 Worked Problem G — breadth-first search, traced

Problem. (a) Give the degree of every vertex of G_1 and verify the handshake lemma. (b) Run breadth-first search from A, giving a table with one row per turn of the **while** loop: the vertex dequeued, what it discovered, and the queue afterward. (c) Tabulate $d[v]$ and $\pi[v]$ and state a shortest path from A to I. (d) How many tree edges are there, and why must it be that number?

What it tests. Whether you can execute the algorithm of **Figure 12.10** (§10.7.3) exactly, including the part people skip: *examining* an edge to a vertex that is already gray, and correctly doing nothing.

```
BFS (G, s)
for each vertex u ∈ V[G] − {s}
    color[u] ← WHITE
    d[u] ← ∞
    π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← {s}
while Q ≠ ∅ do
    u ← head[Q]
    for each v ∈ Adj[u] do
        if color[v] = WHITE then
            color[v] ← GRAY
            d[v] ← d[u] + 1
            π[v] ← u
            ENQUEUE (Q, v)
    DEQUEUE (Q)
    color[u] ← BLACK
```

Initialize every vertex except s

Initialize the source and the queue

Handle all of u's children before handling any children of children

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.

Solution.

(a) Degrees. $\deg(A) = 2$ (B, C), $\deg(B) = 3$ (A, D, E), $\deg(C) = 3$ (A, E, F), $\deg(D) = 2$ (B, G), $\deg(E) = 4$ (B, C, G, H), $\deg(F) = 2$ (C, H), $\deg(G) = 3$ (D, E, I), $\deg(H) = 3$ (E, F, I), $\deg(I) = 2$ (G, H). The handshake lemma (§10.4.3) says

$$\sum_{v \in V} \deg(v) = 2\,|E|, \tag{12.13}$$

because every edge contributes one to the degree of each of its two endpoints. Here the left side of Equation (12.13) is $2 + 3 + 3 + 2 + 4 + 2 + 3 + 3 + 2 = 24$ and the right side is $2 \times 12 = 24$. ✓

(b) The trace. Initialize $d[A] = 0$, everything else ∞, and enqueue A; **Table 12.21** is the result.

Table 12.21 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)

The two rows that matter most are 3 and 5. At step 3, C's list contains E, and E is already gray with $d[E] = 2$; we could compute $d[C] + 1 = 2$ and get the same number, but the algorithm must not touch E, because **first touch wins** and re-touching would change $\pi[E]$. At step 5, E's list contains G, already gray at $d = 3$; ignoring it is what keeps the tree a tree.

(c) The arrays, and the path. **Table 12.22** collects the output.

Table 12.22 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

A shortest path to I: $\pi[I] = G$, $\pi[G] = D$, $\pi[D] = B$, $\pi[B] = A$. Reversed, $A \to B \to D \to G \to I$, of length $4 = d[I]$. ✓ There is a second shortest path, $A\ C\ E\ H\ I$, also of length 4; BFS returns one of them, not all.

(d) Tree edges. Eight — one for every vertex except the source, since each non-source vertex is discovered exactly once and acquires exactly one predecessor. And $8 = 9 - 1 = |V| - 1$, which is the edge count of any tree on nine vertices (§11.4.2). **Figure 12.11** shows the same fact on the eight-vertex example of §10.7.4: eight vertices, seven thick edges.

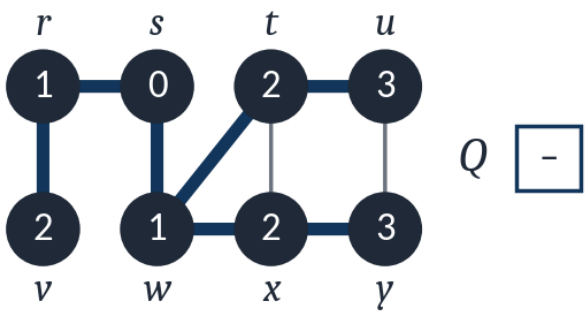


Figure 12.11 A finished breadth-first search: every vertex black, the queue empty, each vertex carrying its distance from s, and the seven thick edges forming the breadth-first tree over eight vertices.

THE TRAP. The *distances* a BFS computes are a property of the graph and do not depend on the order of the adjacency lists. The *tree*, the π array and the queue contents all do. If your π differs from someone else's, you may both be right; if your d differs, at least one of you is wrong.

Answer. Degrees sum to $24 = 2|E|$; $d = (0,1,1,2,2,2,3,3,4)$ for A to I; a shortest path to I is $A\ B\ D\ G\ I$; eight tree edges.

CHECK YOURSELF. Run BFS on G_1 from E instead. Give d and π for every vertex, and the greatest distance from E.

Answer. d: A 2, B 1, C 1, D 2, E 0, F 2, G 1, H 1, I 2. π: $A \leftarrow B$, $B \leftarrow E$, $C \leftarrow E$, $D \leftarrow B$, $F \leftarrow C$, $G \leftarrow E$, $H \leftarrow E$, $I \leftarrow G$. The greatest distance is **2**: every vertex is within two edges of E, which makes E the most central vertex of G_1 (from A the greatest distance was 4).

12.4.3 Worked Problem H — depth-first search of a digraph, every arc classified

Problem. (a) Run depth-first search on G_2 and give $d[v]/f[v]$ for every vertex. (b) Describe the depth-first forest and say how many trees it has. (c) Classify every one of the eleven arcs as tree, back, forward or cross, giving the color test that decided each. (d) Write the parenthesis structure, and say what it tells you about B and C.

What it tests. The color test of **Figure 12.12** (§10.8.2), applied without hesitation eleven times, and the discipline of stamping the clock on *both* events.

```
DFS (G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
        then DFS-VISIT (u)

DFS-VISIT (u)
color[u] ← GRAY                    ▷ White vertex u discovered.
d[u] ← time                        ▷ Mark with discovery time.
time ← time + 1                    ▷ Tick global time.
for each v ∈ Adj[u]                ▷ Explore all edges (u, v).
    do if color[v] = WHITE
        then DFS-VISIT (v)
color[u] ← BLACK                   ▷ Blacken u; it is finished.
f[u] ← time                        ▷ Mark with finishing time.
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 $f[u]$.

Solution.

(a) The trace. Adjacency lists alphabetically: $\text{Adj}[A] = [B, C, D]$, $\text{Adj}[B] = [D]$, $\text{Adj}[C] = [D, E]$, $\text{Adj}[D] = [A]$, $\text{Adj}[E] = [F]$, $\text{Adj}[F] = [E]$, $\text{Adj}[G] = [C, F]$.

- A is white. Visit it: gray, $d[A] = 1$.
- $\text{Adj}[A]$, first B: white. **Tree edge** $A \to B$. Gray, $d[B] = 2$.
- $\text{Adj}[B] = [D]$: white. **Tree edge** $B \to D$. Gray, $d[D] = 3$.
- $\text{Adj}[D] = [A]$: A is **gray**, so it is an ancestor. **Back edge** $D \to A$. This is the algorithm finding the cycle $A \to B \to D \to A$.
- D's list is exhausted: black, $f[D] = 4$. Return to B; exhausted: black, $f[B] = 5$. Return to A.
- $\text{Adj}[A]$, next C: white. **Tree edge** $A \to C$. Gray, $d[C] = 6$.
- $\text{Adj}[C]$, first D: **black**, and $d[C] = 6 > 3 = d[D]$, so D is neither an ancestor nor a descendant of C. **Cross edge** $C \to D$.
- Next E: white. **Tree edge** $C \to E$. Gray, $d[E] = 7$.
- $\text{Adj}[E] = [F]$: white. **Tree edge** $E \to F$. Gray, $d[F] = 8$.
- $\text{Adj}[F] = [E]$: **gray**. **Back edge** $F \to E$ — the two-cycle $E \rightleftarrows F$.
- F black, $f[F] = 9$. E black, $f[E] = 10$. C black, $f[C] = 11$.
- Back in $\text{Adj}[A]$, last entry D: **black**, and $d[A] = 1 < 3 = d[D]$, so D *is* a descendant of A (reached through B). **Forward edge** $A \to D$.

- A black, $f[A] = 12$. The first tree is complete.
- The outer loop finds B, C, D, E, F black and G white. Visit G: gray, $d[G] = 13$. $\text{Adj}[G] = [C, F]$: both black with smaller d, so both are **cross edges**. G black, $f[G] = 14$.

Table 12.23 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	–

(b) The forest. Reading **Table 12.23**, there are two roots, A and G — the two vertices with no predecessor — so the forest has **two trees**: $\{A, B, D, C, E, F\}$ and $\{G\}$. Tree edges number $|V| -$ (number of trees) $= 7 - 2 = 5$, which matches the five found above.

(c) The classification. **Table 12.24** lists every arc with the test that decided it.

Table 12.24 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
$F \to E$	**gray**	**back**
$G \to C$	black, $d[G] > d[C]$	cross
$G \to F$	black, $d[G] > d[F]$	cross

The single rule to memorize is in the middle column: **gray means back, white means tree, and black splits into forward or cross by comparing discovery times.** **Figure 12.13** makes the same classification visual on the larger example of §10.8.10 — back edges curve *up*, forward edges point *down* past a level, cross edges go *sideways*.

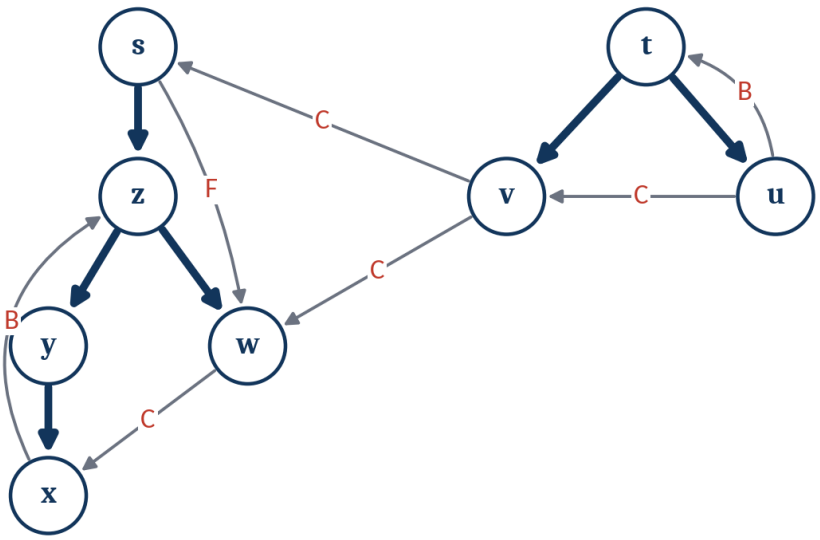


Figure 12.13 A larger example redrawn as its depth-first forest, tree edges thick and downward. Now the classification is geometric: **B** curves back up to an ancestor, **F** points down past an intermediate level to a descendant, and **C** goes sideways between branches or between trees.

(d) The parenthesis structure. Writing "(" at each discovery and ")" at each finish, in clock order:

$$(A\ (B\ (D\ D)\ B)\ (C\ (E\ (F\ F)\ E)\ C)\ A)\ (G\ G)$$

Every pair is either nested inside another or entirely disjoint from it; no two overlap partially. That is the parenthesis theorem (§10.8.8), drawn in **Figure 12.14**.

What it tells us about B and C: their intervals are [2,5] and [6,11], which are **disjoint**. So neither is an ancestor of the other in the forest — they are separate branches below A. Had the intervals been nested, one would be an ancestor of the other; had they overlapped partially, the algorithm would be broken.

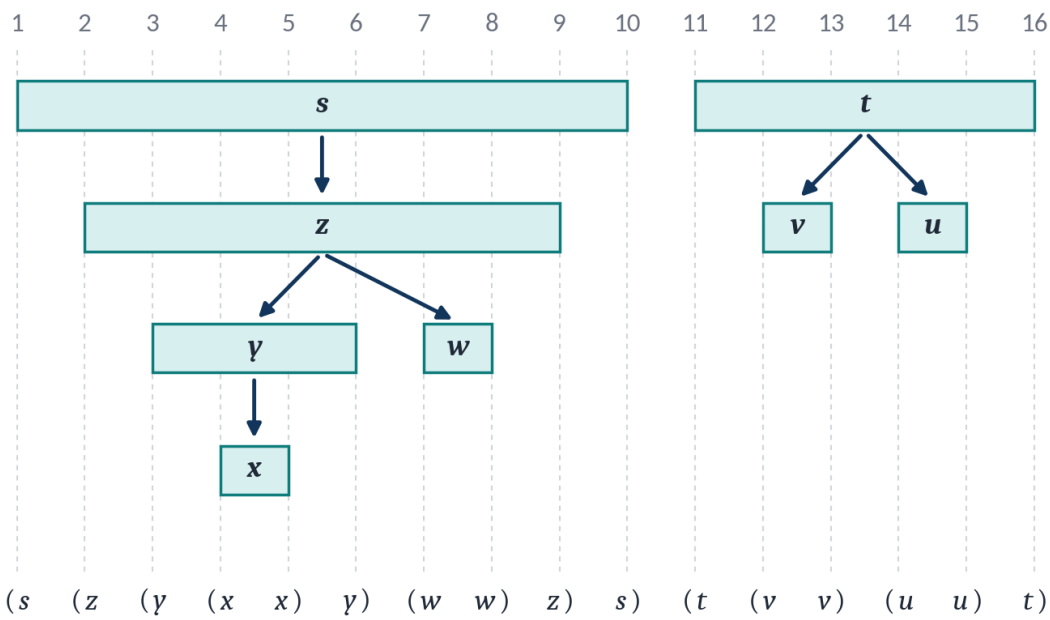


Figure 12.14 The parenthesis structure. Above, the depth-first forest with tree edges pointing downward; below, the clock from 1 to 16 with one bar per vertex spanning $[d, f]$, and the parenthesization written along the bottom. Every pair of bars is nested or disjoint — never partially overlapping.

Figure 12.15 is the fully timestamped example of §10.8.4, and it is worth reproducing your own answer in that form: one box per vertex holding d/f, thick arrows for tree edges, and **B**, **F**, **C** written on the others.

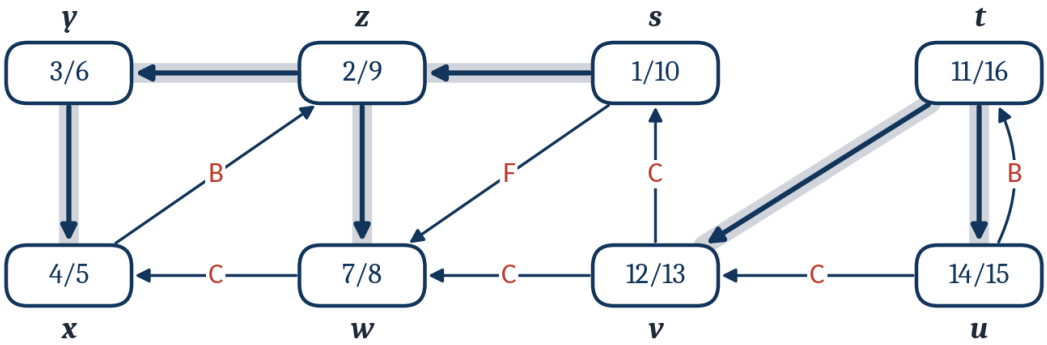


Figure 12.15 An eight-vertex depth-first search, timestamped: s is 1/10, z is 2/9, y is 3/6, x is 4/5, w is 7/8, t is 11/16, v is 12/13, u is 14/15. One figure carrying the timestamps, the forest and the complete edge classification at once — the form to copy when you write out a search of your own.

Answer. d/f as in **Table 12.23**; two trees; five tree, two back, one forward and three cross arcs; B and C have disjoint intervals and are unrelated in the forest.

> **CHECK YOURSELF.** Keep everything the same except that A's list is stored as $[D, C, B]$. Redo parts (a) and (c). Which classes change?
>
> *Answer.* d/f: A 1/12, D 2/3, C 4/9, E 5/8, F 6/7, B 10/11, G 13/14. Now $A \to D$ is a **tree** edge (it is examined first, while D is white), $D \to A$ is still back, $A \to C$, $C \to E$, $E \to F$ and $A \to B$ are tree, $F \to E$ is back, and $C \to D$, $B \to D$, $G \to C$, $G \to F$ are cross. Totals: five tree, two back, **zero** forward, **four** cross. The forward edge has vanished and a cross edge has appeared; the verdict "not acyclic" is unchanged, because it is a property of the graph and not of the search.

12.4.4 Worked Problem I — depth-first search of an undirected graph

Problem. Run depth-first search on the *undirected* graph G_1, starting at A. (a) Give a table of events in clock order — every discovery, every finish, and every edge that reaches a gray vertex other than the parent. (b) Give d/f and π for every vertex. (c) Classify every edge, and explain why only two of the four classes appear. (d) Relate the number of back edges to the number of independent cycles of G_1.

What it tests. Depth-first search on an undirected graph (§10.8.10), where every edge sits in *two* adjacency lists and is therefore examined twice, and the rule that decides what a second sighting means.

Solution.

(a) The events. The adjacency lists, alphabetically, are A: $[B, C]$, B: $[A, D, E]$, C: $[A, E, F]$, D: $[B, G]$, E: $[B, C, G, H]$, F: $[C, H]$, G: $[D, E, I]$, H: $[E, F, I]$, I: $[G, H]$. When the search at u examines a neighbor v there are four cases:

- v **white**: a tree edge; recurse.
- v **gray** and v is u's parent: this is the tree edge $\{u, v\}$ seen again from below. Skip it.
- v **gray** and not the parent: v is a proper ancestor, so $\{u, v\}$ is a **back edge**.
- v **black**: v was discovered after u and has already finished, so v is a descendant of u and the edge was already classified as a back edge *from v's side*. Skip it.

Following the lists:

- Discover A ($t = 1$). First neighbor B is white: discover B ($t = 2$).
- At B: A is the parent. D is white: discover D ($t = 3$).
- At D: B is the parent. G is white: discover G ($t = 4$).
- At G: D is the parent. E is white: discover E ($t = 5$).
- At E: B is **gray and not the parent** (the parent is G): **back edge** EB. C is white: discover C ($t = 6$).
- At C: A is gray and not the parent: **back edge** CA. E is the parent. F is white: discover F ($t = 7$).
- At F: C is the parent. H is white: discover H ($t = 8$).
- At H: E is gray and not the parent: **back edge** HE. F is the parent. I is white: discover I ($t = 9$).
- At I: G is gray and not the parent: **back edge** IG. H is the parent. The list is exhausted: finish I ($t = 10$).
- Unwinding: H has nothing left, finish ($t = 11$); F finishes ($t = 12$); C finishes ($t = 13$).
- Back at E: G is the parent; H is **black** — the edge HE, already classified from H's side. Finish E ($t = 14$).
- Back at G: I is black (the edge IG, already classified). Finish G ($t = 15$). Then D ($t = 16$).
- Back at B: E is black (the edge EB). Finish B ($t = 17$). Back at A: C is black (the edge CA). Finish A ($t = 18$).

(b) The table. **Table 12.25** collects the stamps and predecessors.

Table 12.25 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

(c) The classification. Tree edges: $AB, BD, DG, GE, EC, CF, FH, HI$ — eight, which is $|V| - 1$ because the graph is connected and the forest is a single tree. Back edges: EB, CA, HE, IG — four. Together they account for all $8 + 4 = 12$ edges.

Forward and cross edges **cannot** occur in an undirected graph. A forward edge would be an edge $\{u, v\}$ first examined from u when v is already a finished descendant — but v, while it was being searched, had u in its own list and found u gray, so it classified the edge as a back edge first. A cross edge would join two vertices neither of which is an ancestor of the other — but whichever of the two was discovered first would have found the other still white and made the edge a tree edge. This is the theorem of §10.8.10, and the trace shows it in action: every black sighting was an edge already classified from its other end.

(d) Back edges and cycles. G_1 is connected, so its number of independent cycles is $|E| - |V| + 1 = 12 - 9 + 1 = 4$ — the circuit rank of §12.4.5. The search found exactly **4** back edges. That is not a coincidence: every edge not in the depth-first tree is a back edge, and every non-tree edge closes exactly one cycle with the tree. The four cycles are $E\,B\,D\,G\,E$, $C\,A\,B\,D\,G\,E\,C$, $H\,E\,C\,F\,H$ and $I\,G\,E\,C\,F\,H\,I$ — each a back edge plus the tree path between its endpoints.

Listing 12.6 prints the whole event table and the stamps, and runs the search a second time from E for the check-yourself below.

Listing 12.6 Depth-first search of an undirected graph, printing every discovery, every finish and every back edge in clock order, then the d/f table, both edge classes and the parenthesis structure; run from A and from E. Compile with `g++ -std=c++17 -O2 -Wall -o dfsu dfsu.cpp`.

```
// Worked Problem I: depth-first search of an UNDIRECTED graph, with the
// discovery/finish table, every edge classified, and the parenthesis structure.
// The graph is G1 of Problem G: vertices A..I, adjacency lists alphabetical.
// In an undirected graph only tree edges and back edges can occur.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;

static const string NM = "ABCDEFGHI";
static const vector<pair<int,int>> EDGES = {
    {0,1},{0,2},{1,3},{1,4},{2,4},{2,5},{3,6},{4,6},{4,7},{5,7},{6,8},{7,8}
};

string stamp(int t) { string s = "t=" + to_string(t); return s + string(6 - s.size(), ' ');
}

void run(int start) {
    int n = (int)NM.size();
    vector<vector<int>> adj(n);
    for (auto [a, b] : EDGES) { adj[a].push_back(b); adj[b].push_back(a); }
    for (auto& l : adj) sort(l.begin(), l.end());
    vector<int> col(n, 0), d(n, 0), f(n, 0), pi(n, -1);      // 0 white, 1 gray, 2 black
    vector<string> events;                                    // the trace, one line per
event
    vector<string> tree, back;
    int clk = 0;
    function<void(int)> visit = [&](int u) {
        col[u] = 1; d[u] = ++clk;
        events.push_back(stamp(clk) + "discover " + NM[u]);
        for (int v : adj[u]) {
            string e = string(1, NM[u]) + NM[v];
            if (col[v] == 0) {
                tree.push_back(e); pi[v] = u;
                visit(v);
            } else if (col[v] == 1 && v != pi[u]) {
                back.push_back(e);                            // gray and not the parent
                events.push_back("         " + e + " finds " + NM[v] + " gray: back edge");
            }
            // gray parent: the tree edge seen from below; black: the back edge seen
            // from its other end, already classified. Neither is a new edge.
        }
        col[u] = 2; f[u] = ++clk;
        events.push_back(stamp(clk) + "finish   " + NM[u]);
    };
```

Listing 12.6 (Continued) *Depth-first search of an undirected graph, printing every discovery, every finish and every back edge in clock order, then the d/f table, both edge classes and the parenthesis structure; run from A and from E. Compile with g++ -std=c++17 -O2 -Wall -o dfsu dfsu.cpp.*

```
    visit(start);
    for (int v = 0; v < n; ++v) if (col[v] == 0) visit(v);
    cout << "=== DFS of G1 starting at " << NM[start] << " ===\n";
    for (const string& s : events) cout << "  " << s << "\n";
    cout << "  v     "; for (int v = 0; v < n; ++v) cout << setw(6) << NM[v]; cout << "\n";
    cout << "  d/f   "; for (int v = 0; v < n; ++v) cout << setw(6) << (to_string(d[v]) +
"/" + to_string(f[v])); cout << "\n";
    cout << "  pi    "; for (int v = 0; v < n; ++v) cout << setw(6) << (pi[v] < 0 ?
string("-") : string(1, NM[pi[v]])); cout << "\n";
    cout << "  tree edges (" << tree.size() << "):"; for (auto& e : tree) cout << " " << e;
cout << "\n";
    cout << "  back edges (" << back.size() << "):"; for (auto& e : back) cout << " " << e;
cout << "\n";
    vector<pair<int,int>> ev;
    for (int v = 0; v < n; ++v) { ev.push_back({d[v], v}); ev.push_back({f[v], ~v}); }
    sort(ev.begin(), ev.end());
    cout << "  parentheses: ";
    for (auto [t, x] : ev) cout << (x >= 0 ? "(" + string(1, NM[x]) : string(1, NM[~x]) +
")") << " ";
    cout << "\n";
}

int main() { run(0); cout << "\n"; run(4); return 0; }
```

```
=== DFS of G1 starting at A ===
  t=1   discover A
  t=2   discover B
  t=3   discover D
  t=4   discover G
  t=5   discover E
           EB finds B gray: back edge
  t=6   discover C
           CA finds A gray: back edge
  t=7   discover F
  t=8   discover H
           HE finds E gray: back edge
  t=9   discover I
           IG finds G gray: back edge
  t=10  finish   I
  t=11  finish   H
  t=12  finish   F
  t=13  finish   C
  t=14  finish   E
  t=15  finish   G
  t=16  finish   D
  t=17  finish   B
  t=18  finish   A
  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
  pi         -     A     E     B     G     C     D     F     H
  tree edges (8): AB BD DG GE EC CF FH HI
  back edges (4): EB CA HE IG
  parentheses: (A (B (D (G (E (C (F (H (I I) H) F) C) E) G) D) B) A)

=== DFS of G1 starting at E ===
  t=1   discover E
  t=2   discover B
  t=3   discover A
  t=4   discover C
           CE finds E gray: back edge
  t=5   discover F
  t=6   discover H
           HE finds E gray: back edge
  t=7   discover I
  t=8   discover G
  t=9   discover D
           DB finds B gray: back edge
  t=10  finish   D
           GE finds E gray: back edge
  t=11  finish   G
  t=12  finish   I
  t=13  finish   H
  t=14  finish   F
  t=15  finish   C
  t=16  finish   A
  t=17  finish   B
  t=18  finish   E
  v          A     B     C     D     E     F     G     H     I
  d/f     3/16  2/17  4/15  9/10  1/18  5/14  8/11  6/13  7/12
  pi         B     E     A     G     -     C     I     F     H
  tree edges (8): EB BA AC CF FH HI IG GD
  back edges (4): CE HE DB GE
  parentheses: (E (B (A (C (F (H (I (G (D D) G) I) H) F) C) A) B) E)
```

Answer. d/f and π as in **Table 12.25**; eight tree edges and four back edges, no forward or cross edges; four back edges for a circuit rank of four.

> **CHECK YOURSELF.** Start the search at E instead of A. Give the d/f stamps and the back edges.

Answer. E 1/18, B 2/17, A 3/16, C 4/15, F 5/14, H 6/13, I 7/12, G 8/11, D 9/10. The tree is again a single path, $E\ B\ A\ C\ F\ H\ I\ G\ D$, and the back edges are CE, HE, DB and GE — a different four, but still four, as the circuit rank demands. The second half of the output of **Listing 12.6** confirms every value.

12.4.5 Worked Problem J — connected components

Problem. G_3 is undirected on the twelve vertices 1, … ,12, with the eleven edges

$$\{1,2\},\{2,3\},\{3,1\},\quad \{4,5\},\{4,6\},\{4,7\},\quad \{8,9\},\{9,10\},\{10,11\},\{11,12\},\{12,8$$

(a) Compute the connected components and give the component label of every vertex. (b) Answer: are 1 and 3 connected? 1 and 4? 5 and 7? 6 and 11? 8 and 12? (c) A forest on 12 vertices with k components has how many edges? What does comparing that with 11 tell you?

What it tests. That "is u connected to v?" should be turned from a search into an array lookup (§10.9.1).

Solution.

(a) The components. Run a breadth-first search from every vertex not yet labeled, giving each search a fresh label.

- Start at 1. Reaches 2 and 3 (the triangle). Component **0** $=\{1,2,3\}$.
- 4 is unlabeled. Start there. Reaches 5, 6, 7 (the star). Component **1** $=\{4,5,6,7\}$.
- 8 is unlabeled. Start there. Reaches 9, 10, 11, 12 around the five-cycle. Component **2** $=\{8,9,10,11,12\}$.

Table 12.26 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

The total cost is $\Theta(V+E)$ for the *whole* array, not per query, because the searches partition the edge set: no edge is scanned by two different searches.

(b) The queries. Read them straight off **Table 12.26**. 1 and 3: both label 0, **yes**. 1 and 4: 0 against 1, **no**. 5 and 7: both 1, **yes**. 6 and 11: 1 against 2, **no**. 8 and 12: both 2, **yes**.

(c) The edge count. A forest on n vertices with k components has exactly $n-k$ edges, because each component is a tree with (its own vertex count -1) edges, and the counts add. Here $12-3=9$. The graph has **11** edges, which is 2 more than a forest would need. So the graph contains exactly $11-9=2$ independent cycles — and indeed we can see them: the triangle 1–2–3 and the five-cycle 8–9–10–11–12. The star is a tree and contributes none.

THE CYCLE-COUNT FORMULA, DERIVED IN ONE LINE. For any undirected graph with k components, the number of independent cycles — its **circuit rank** — is

$$r(G)=|E|-|V|+k.$$

(12.14)

Take any spanning forest: it has $|V|-k$ edges and no cycle. Every one of the remaining $|E|-(|V|-k)$ edges closes exactly one new cycle when added back. By Equation (12.14), $r(G_3)=11-12+3=2$.

Answer. Components $\{1,2,3\}$, $\{4,5,6,7\}$, $\{8,\dots,12\}$; the queries answer yes, no, yes, no, yes; a forest would have 9 edges, so there are 2 independent cycles.

> **CHECK YOURSELF.** Add the edge $\{3,4\}$ to G_3. How many components are there now, and how many independent cycles?
>
> *Answer.* Two components, $\{1,\dots,7\}$ and $\{8,\dots,12\}$. By Equation (12.14), $r = 12 - 12 + 2 = 2$ — **unchanged**. The new edge joined two components instead of closing a cycle: $|E|$ and k moved together. An edge whose removal disconnects its component is called a *bridge*, and $\{3,4\}$ is one.

12.4.6 Worked Problem K — bipartiteness, and its certificate

Problem. (a) Decide whether G_1 is bipartite. If it is, give the two sides. (b) Now add the single edge EF to G_1. Decide again, and if the answer changes, exhibit an odd cycle as the certificate — both the one a breadth-first two-coloring would report and the shortest one in the graph.

What it tests. That bipartite means *two-colorable*, that the test is one line inside a breadth-first search, and that a **negative** answer must come with a certificate (§10.9.3).

Solution.

(a) The test on G_1. Color A with 0 and give every vertex the opposite color to its discoverer. From **Table 12.22** the distances are A: 0, B: 1, C: 1, D: 2, E: 2, F: 2, G: 3, H: 3, I: 4, so the color is simply $d[v]$ mod 2:

- color 0 (even distance): A, D, E, F, I;
- color 1 (odd distance): B, C, G, H.

Now check all twelve edges, which is the part that cannot be skipped: $AB(0,1)$ ✓, $AC(0,1)$ ✓, $BD(1,0)$ ✓, $BE(1,0)$ ✓, $CE(1,0)$ ✓, $CF(1,0)$ ✓, $DG(0,1)$ ✓, $EG(0,1)$ ✓, $EH(0,1)$ ✓, $FH(0,1)$ ✓, $GI(1,0)$ ✓, $HI(1,0)$ ✓. Every edge joins the two sides, so G_1 **is bipartite**, with sides $\{A, D, E, F, I\}$ and $\{B, C, G, H\}$.

(b) Adding EF. E and F both have color 0 — both sit at distance 2 from A. The new edge joins two vertices of the same side, so no two-coloring survives and the graph is **not bipartite**.

The certificate is an odd cycle, and there are two worth giving. A breadth-first two-coloring reports the cycle built from the two tree paths to the offending edge's endpoints: from E up to B to A, down to C, down to F, and back along the new edge —

$$E - B - A - C - F - E, \qquad \text{length } 5.$$

The *shortest* odd cycle in the graph is shorter: E and F are both adjacent to C, so

$$E - C - F - E, \qquad \text{length } 3.$$

Either is a complete answer, because the theorem is *a graph is bipartite if and only if it contains no odd cycle*; exhibiting one odd cycle of any length settles it. A five-cycle is a perfectly good certificate and is what the algorithm naturally produces, since BFS knows the tree, not the whole graph.

Why odd cycles are exactly the obstruction. Walk round a cycle alternating colors. After an even number of steps you are back to the color you started with; after an odd number you are not. A cycle of odd length therefore forces its first vertex to be both colors. Conversely, if no odd cycle

exists, coloring by the parity of the BFS distance works: by Equation (12.4) every edge joins vertices whose distances differ by at most one, and it cannot join two vertices of *equal* distance, or the two tree paths plus that edge would form an odd cycle.

Answer. G_1 is bipartite with sides $\{A, D, E, F, I\}$ and $\{B, C, G, H\}$; with EF added it is not, certified by $E\ B\ A\ C\ F\ E$ (length 5) or $E\ C\ F\ E$ (length 3).

> **CHECK YOURSELF.** Instead of EF, add (a) the edge AI; (b) the edge AH. Is G_1 still bipartite in each case?
>
> *Answer.* (a) **No.** $d[A] = 0$ and $d[I] = 4$ are both even, so A and I have the same color; an odd cycle is $A\ B\ D\ G\ I\ A$, of length 5. (b) **Yes.** $d[H] = 3$ is odd, so A and H are on opposite sides and the same two sides still work.

12.4.7 Worked Problem L — cycle detection, undirected and directed

Problem. (a) State the rule for detecting a cycle in an *undirected* graph by depth-first search, and explain precisely why "I have seen this vertex before" is not the rule. (b) Apply your rule to the path A–B–C–D, to the square A–B–C–D–A, and to G_1. (c) State the rule for a *directed* graph and apply it to G_2 and to the digraph $A \to B,\ A \to C,\ B \to C$.

What it tests. The two cycle tests of §10.9.2 and §11.2, and the difference between them.

Solution.

(a) The undirected rule, and why the obvious one is wrong. The obvious rule — *"during a depth-first search, if you reach a vertex you have already seen, there is a cycle"* — is **false for undirected graphs**, and the smallest counterexample has two vertices. In the graph A–B, the search visits A, walks the edge to B, and then examines $\mathrm{Adj}[B] = [A]$ and sees A, already gray. But there is no cycle: the algorithm is merely looking back along the edge it just arrived on. In an undirected graph every edge appears in *both* adjacency lists, so this happens at every single vertex.

The correct rule is:

> **UNDIRECTED CYCLE TEST.** During a depth-first search from u with parent p, examine each $v \in \mathrm{Adj}[u]$. Skip **exactly one** occurrence of p — the edge you came in on. If any other non-white vertex is found, a cycle exists.

The words "exactly one" matter. If the graph has two parallel edges between A and B, skipping *every* occurrence of the parent would miss the cycle those two edges form. Skipping the first and then testing the rest catches it.

(b) Applied.

- **Path A–B–C–D.** Visit A, B (parent A), C (parent B), D (parent C). At each step the only non-white neighbor is the parent, skipped once. No cycle: **acyclic**, and with $|E| = 3 = |V| - 1$ and one component it is a tree.
- **Square A–B–C–D–A.** Visit A, B, C, D; $\mathrm{Adj}[D] = [A, C]$; C is the parent, skipped; A is gray and is not the parent. **Cycle found.** Here $|E| = 4 > 3 = |V| - 1$, so Equation (12.14) had already guaranteed one.
- **G_1.** $|V| = 9$, $|E| = 12$, one component, so the circuit rank is $12 - 9 + 1 = 4$: there are four independent cycles, and Problem I found all four back edges. **Has a cycle.**

(c) The directed rule. In a directed graph the far endpoint's color is the whole test, and **only gray counts**:

DIRECTED CYCLE TEST. An arc $u \to v$ examined while v is **gray** is a back edge, and a directed graph has a directed cycle if and only if a depth-first search of it produces at least one back edge (§11.2.3). A **black** v means v's whole search finished before this arc was examined, so v cannot reach u, and the arc closes nothing.

Applied to G_2: **Table 12.24** found two back edges, $D \to A$ and $F \to E$, so G_2 **has directed cycles** — namely $A \to B \to D \to A$ and $E \to F \to E$. Applied to $A \to B$, $A \to C$, $B \to C$: visiting A then B then C, the arc $B \to C$ is a tree edge and $A \to C$ finds C black with $d[A] < d[C]$ — a *forward* edge, not a back edge. Zero back edges, so this one **is a DAG**, even though C is "seen twice".

THE TRAP. Applying the *directed* test (any non-white neighbor means a cycle) to an undirected graph reports a cycle in every graph that has an edge. Applying the *undirected* test (skip the parent) to a directed graph is meaningless, because a directed graph has no "edge you came in on" to skip. The two tests are not interchangeable.

Answer. Undirected: skip the parent exactly once, then any non-white neighbor means a cycle — the path is acyclic, the square and G_1 are not. Directed: a gray neighbor means a cycle — G_2 has two, and the three-arc digraph is a DAG.

CHECK YOURSELF. Decide each: (a) the undirected graph with edges AB, BC, CA, CD; (b) the digraph $A \to B, B \to C, C \to D, D \to B$; (c) the digraph $A \to B, A \to C, B \to D, C \to D$.

Answer. (a) **Cycle**: at C (parent B), the neighbor A is gray and not the parent. (b) **Cycle**: the search goes A, B, C, D, and at D the arc $D \to B$ finds B gray — the cycle $B \to C \to D \to B$. (c) **DAG**: D is reached from B as a tree edge, and later from C it is black with $d[C] > d[D]$, a cross edge; no arc ever meets a gray vertex.

12.4.8 Problems G, H, J, K and L as one program

Listing 12.7 runs Problems G, H, J, K and L. The listing is complete; the output below it is everything it printed.

Listing 12.7 Worked Problems G, H, J, K and L, computed: the BFS trace, the DFS timestamps and classification, the components, the two-coloring with its odd-cycle certificate, and both cycle tests. Compile with `g++ -std=c++17 -O2 -Wall -o search search.cpp`.

```
// Worked Problems G, H, J, K and L of Chapter 12 -- searching a graph.
//   G  breadth-first search: the trace, d, pi, and the tree
//   H  depth-first search: timestamps, forest, and every arc classified
//   J  connected components
//   K  bipartiteness, and the odd cycle that destroys it
//   L  cycle detection, undirected and directed
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
#include <functional>
using namespace std;

static const string UN = "ABCDEFGHI";              // the undirected graph G1 of G and K
static const vector<pair<int,int>> UE = {
    {0,1},{0,2},{1,3},{1,4},{2,4},{2,5},{3,6},{4,6},{4,7},{5,7},{6,8},{7,8}
};

// ---------------------------------------------------------------- G
void q1() {
    cout << "=== Problem G  breadth-first search from A ===\n";
    int n = (int)UN.size();
    vector<vector<int>> adj(n);
    for (auto [a,b] : UE) { adj[a].push_back(b); adj[b].push_back(a); }
    for (auto& l : adj) sort(l.begin(), l.end());          // alphabetical, as the problem
says
    vector<int> d(n, INT_MAX), pi(n, -1);
    d[0] = 0; queue<int> Q; Q.push(0);
    cout << "step  dequeue  discovered            queue after\n";
    int step = 0;
    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        string disc;
        for (int v : adj[u]) if (d[v] == INT_MAX) {
            d[v] = d[u] + 1; pi[v] = u; Q.push(v);
            disc += string(1, UN[v]) + ":" + to_string(d[v]) + " ";
        }
        queue<int> tmp = Q; string qs;
        while (!tmp.empty()) { qs += string(1, UN[tmp.front()]) + ":" +
to_string(d[tmp.front()]) + " "; tmp.pop(); }
        cout << setw(4) << ++step << setw(9) << UN[u] << "  " << left << setw(22)
             << (disc.empty() ? "-" : disc) << right << (qs.empty() ? "(empty)" : qs) <<
"\n";
    }
    cout << "v      "; for (int v = 0; v < n; ++v) cout << setw(4) << UN[v]; cout << "\n";
    cout << "d[v]   "; for (int v = 0; v < n; ++v) cout << setw(4) << d[v];  cout << "\n";
```

Listing 12.7 (Continued) *Worked Problems G, H, J, K and L, computed: the BFS trace, the DFS timestamps and classification, the components, the two-coloring with its odd-cycle certificate, and both cycle tests. Compile with g++ -std=c++17 -O2 -Wall -o search search.cpp.*

```
    cout << "pi[v]  "; for (int v = 0; v < n; ++v) cout << setw(4) << (pi[v] < 0 ?
string("-") : string(1, UN[pi[v]])); cout << "\n";
    int tree = 0; for (int v = 0; v < n; ++v) if (pi[v] != -1) ++tree;
    cout << "tree edges: " << tree << " = |V| - 1 = " << n - 1 << "\n";
}

// ---------------------------------------------------------------- H
static const string DN = "ABCDEFG";                 // the directed graph G2 of H and L
static const vector<pair<int,int>> DE = {
    {0,1},{0,2},{0,3},{1,3},{2,3},{2,4},{3,0},{4,5},{5,4},{6,2},{6,5}
};
void q2() {
    cout << "\n=== Problem H  depth-first search, all seven vertices ===\n";
    int n = (int)DN.size();
    vector<vector<int>> adj(n);
    for (auto [u,v] : DE) adj[u].push_back(v);
    for (auto& l : adj) sort(l.begin(), l.end());
    vector<int> color(n, 0), d(n, 0), f(n, 0), pi(n, -1);   // 0 white 1 gray 2 black
    vector<string> cls(DE.size());
    int clock_ = 0;
    function<void(int)> visit = [&](int u) {
        color[u] = 1; d[u] = ++clock_;
        for (int v : adj[u]) {
            int idx = -1;
            for (size_t i = 0; i < DE.size(); ++i) if (DE[i].first == u && DE[i].second ==
v) idx = (int)i;
            if (color[v] == 0)      { cls[idx] = "tree";    pi[v] = u; visit(v); }
            else if (color[v] == 1) { cls[idx] = "back"; }
            else                     { cls[idx] = (d[u] < d[v] ? "forward" : "cross"); }
        }
        color[u] = 2; f[u] = ++clock_;
    };
    for (int v = 0; v < n; ++v) if (color[v] == 0) visit(v);
    cout << "v       "; for (int v = 0; v < n; ++v) cout << setw(6) << DN[v]; cout << "\n";
    cout << "d/f     "; for (int v = 0; v < n; ++v) cout << setw(6) << (to_string(d[v]) +
"/" + to_string(f[v])); cout << "\n";
    cout << "pi      "; for (int v = 0; v < n; ++v) cout << setw(6) << (pi[v] < 0 ?
string("-") : string(1, DN[pi[v]])); cout << "\n";
    cout << "clock stopped at " << clock_ << " = 2|V| = " << 2 * n << "\n";
    int t = 0, b = 0, fw = 0, c = 0;
    cout << "arc     class\n";
    for (size_t i = 0; i < DE.size(); ++i) {
        cout << " " << DN[DE[i].first] << "->" << DN[DE[i].second] << "   " << cls[i] <<
"\n";
        if (cls[i] == "tree") ++t; else if (cls[i] == "back") ++b;
        else if (cls[i] == "forward") ++fw; else ++c;
    }
    cout << "tree " << t << ", back " << b << ", forward " << fw << ", cross " << c
         << "  (total " << DE.size() << ")\n";
    int roots = 0; for (int v = 0; v < n; ++v) if (pi[v] == -1) ++roots;
    cout << "the forest has " << roots << " trees, so " << n << " - " << roots
```

Listing 12.7 (Continued) *Worked Problems G, H, J, K and L, computed: the BFS trace, the DFS timestamps and classification, the components, the two-coloring with its odd-cycle certificate, and both cycle tests. Compile with g++ -std=c++17 -O2 -Wall -o search search.cpp.*

```
        << " = " << n - roots << " tree edges\n";
    vector<pair<int,int>> ev;                                   // the parenthesization
    for (int v = 0; v < n; ++v) { ev.push_back({d[v], v}); ev.push_back({f[v], ~v}); }
    sort(ev.begin(), ev.end());
    cout << "parenthesis structure: ";
    for (auto [time, x] : ev) cout << (x >= 0 ? "(" + string(1, DN[x]) : string(1, DN[~x])
+ ")") << " ";
    cout << "\n";
    cout << "the graph has " << b << " back edge(s), so it is "
         << (b ? "NOT acyclic" : "acyclic") << "\n";
}

// ---------------------------------------------------------------- J
void q3() {
    cout << "\n=== Problem J  connected components ===\n";
    int n = 12;
    vector<pair<int,int>> E = {
        {0,1},{1,2},{2,0},                                      // a triangle
        {3,4},{3,5},{3,6},                                      // a star
        {7,8},{8,9},{9,10},{10,11},{11,7}                       // a five-cycle
    };
    vector<vector<int>> adj(n);
    for (auto [a,b] : E) { adj[a].push_back(b); adj[b].push_back(a); }
    vector<int> comp(n, -1); int nc = 0;
    for (int v = 0; v < n; ++v) if (comp[v] == -1) {
        queue<int> Q; Q.push(v); comp[v] = nc;
        while (!Q.empty()) { int u = Q.front(); Q.pop();
            for (int w : adj[u]) if (comp[w] == -1) { comp[w] = nc; Q.push(w); } }
        ++nc;
    }
    cout << "vertex  "; for (int v = 0; v < n; ++v) cout << setw(4) << v + 1; cout << "\n";
    cout << "comp    "; for (int v = 0; v < n; ++v) cout << setw(4) << comp[v]; cout <<
"\n";
    cout << "number of components: " << nc << "\n";
    auto same = [&](int a, int b) { return comp[a-1] == comp[b-1] ? "yes" : "no"; };
    cout << "  1 and 3  connected? " << same(1,3)  << "\n";
    cout << "  1 and 4  connected? " << same(1,4)  << "\n";
    cout << "  5 and 7  connected? " << same(5,7)  << "\n";
    cout << "  6 and 11 connected? " << same(6,11) << "\n";
    cout << "  8 and 12 connected? " << same(8,12) << "\n";
    cout << "edges " << E.size() << ", vertices " << n << ", components " << nc
         << "; a forest would need " << n - nc << " edges, this has " << E.size() << "\n";
}
```

Listing 12.7 (Continued) *Worked Problems G, H, J, K and L, computed: the BFS trace, the DFS timestamps and classification, the components, the two-coloring with its odd-cycle certificate, and both cycle tests. Compile with g++ -std=c++17 -O2 -Wall -o search search.cpp.*

```
// ---------------------------------------------------------------- K
void q4() {
    cout << "\n=== Problem K  bipartiteness ===\n";
    auto test = [](int n, vector<pair<int,int>> E, const string& name, const string& lbl) {
        vector<vector<int>> adj(n);
        for (auto [a,b] : E) { adj[a].push_back(b); adj[b].push_back(a); }
        for (auto& l : adj) sort(l.begin(), l.end());
        vector<int> col(n, -1), pi(n, -1);
        cout << name << ": ";
        for (int s = 0; s < n; ++s) if (col[s] == -1) {
            col[s] = 0; queue<int> Q; Q.push(s);
            while (!Q.empty()) { int u = Q.front(); Q.pop();
                for (int v : adj[u]) {
                    if (col[v] == -1) { col[v] = 1 - col[u]; pi[v] = u; Q.push(v); }
                    else if (col[v] == col[u]) {
                        // walk both endpoints up to their meeting point: the odd cycle
                        vector<int> pu, pv; for (int x = u; x != -1; x = pi[x])
pu.push_back(x);
                        for (int x = v; x != -1; x = pi[x]) pv.push_back(x);
                        reverse(pu.begin(), pu.end()); reverse(pv.begin(), pv.end());
                        size_t k = 0; while (k < pu.size() && k < pv.size() && pu[k] ==
pv[k]) ++k;
                        string cyc; for (size_t i = pu.size(); i-- >= k && i < pu.size();)
cyc += lbl[pu[i]];
                        for (size_t i = k; i < pv.size(); ++i) cyc += lbl[pv[i]];
                        cyc += lbl[u];
                        cout << "NOT bipartite; the edge " << lbl[u] << "-" << lbl[v]
                             << " joins two vertices of the same color. Odd cycle: " << cyc
<< "\n";
                        return;
                    }
                }
            }
        }
        string s0, s1;
        for (int v = 0; v < n; ++v) (col[v] == 0 ? s0 : s1) += lbl[v];
        cout << "BIPARTITE; sides {" << s0 << "} and {" << s1 << "}\n";
    };
    test(9, UE, "G1", UN);
    vector<pair<int,int>> plus = UE; plus.push_back({4,5});     // add E-F
    test(9, plus, "G1 plus the edge E-F", UN);
}
```

Listing 12.7 (Continued) *Worked Problems G, H, J, K and L, computed: the BFS trace, the DFS timestamps and classification, the components, the two-coloring with its odd-cycle certificate, and both cycle tests. Compile with g++ -std=c++17 -O2 -Wall -o search search.cpp.*

```
// ---------------------------------------------------------------- L
void q5() {
    cout << "\n=== Problem L  cycle detection ===\n";
    // undirected: the parent trap
    auto undirected = [](int n, vector<pair<int,int>> E, const string& lbl) {
        vector<vector<int>> adj(n);
        for (auto [a,b] : E) { adj[a].push_back(b); adj[b].push_back(a); }
        vector<int> col(n, 0);
        function<bool(int,int)> go = [&](int u, int parent) {
            col[u] = 1;
            for (int v : adj[u]) {
                if (v == parent) { parent = -2; continue; }   // skip the edge we came in
on, ONCE
                if (col[v]) return true;
                if (go(v, u)) return true;
            }
            col[u] = 2; return false;
        };
        bool cyc = false;
        for (int v = 0; v < n; ++v) if (!col[v] && go(v, -1)) { cyc = true; break; }
        cout << "  " << lbl << ": " << (cyc ? "has a cycle" : "acyclic (a forest)") <<
"\n";
    };
    undirected(4, {{0,1},{1,2},{2,3}}, "path A-B-C-D          ");
    undirected(4, {{0,1},{1,2},{2,3},{3,0}}, "square A-B-C-D-A      ");
    undirected(9, UE, "G1                        ");
    undirected(2, {{0,1},{0,1}}, "two parallel edges A=B");
    // directed: only GRAY counts
    auto directed = [](int n, vector<pair<int,int>> E, const string& lbl) {
        vector<vector<int>> adj(n);
        for (auto [u,v] : E) adj[u].push_back(v);
        vector<int> col(n, 0);
        function<bool(int)> go = [&](int u) {
            col[u] = 1;
            for (int v : adj[u]) { if (col[v] == 1) return true;
                                   if (col[v] == 0 && go(v)) return true; }
            col[u] = 2; return false;
        };
        bool cyc = false;
        for (int v = 0; v < n; ++v) if (!col[v] && go(v)) { cyc = true; break; }
        cout << "  " << lbl << ": " << (cyc ? "has a directed cycle" : "is a DAG") << "\n";
    };
    directed(7, DE, "G2                      ");
    directed(3, {{0,1},{0,2},{1,2}}, "A->B, A->C, B->C       ");
    directed(3, {{0,1},{1,2},{2,0}}, "A->B, B->C, C->A       ");
}
```

Listing 12.7 (Continued) *Worked Problems G, H, J, K and L, computed: the BFS trace, the DFS timestamps and classification, the components, the two-coloring with its odd-cycle certificate, and both cycle tests. Compile with g++ -std=c++17 -O2 -Wall -o search search.cpp.*

```
int main() { q1(); q2(); q3(); q4(); q5(); return 0; }
```

```
=== Problem G  breadth-first search from A ===
step  dequeue  discovered            queue after
   1        A  B:1 C:1               B:1 C:1
   2        B  D:2 E:2               C:1 D:2 E:2
   3        C  F:2                   D:2 E:2 F:2
   4        D  G:3                   E:2 F:2 G:3
   5        E  H:3                   F:2 G:3 H:3
   6        F  -                     G:3 H:3
   7        G  I:4                   H:3 I:4
   8        H  -                     I:4
   9        I  -                     (empty)
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
tree edges: 8 = |V| - 1 = 8

=== Problem H  depth-first search, all seven vertices ===
v            A     B     C     D     E     F     G
d/f       1/12   2/5  6/11   3/4  7/10   8/9 13/14
pi           -     A     A     B     C     E     -
clock stopped at 14 = 2|V| = 14
arc     class
 A->B   tree
 A->C   tree
 A->D   forward
 B->D   tree
 C->D   cross
 C->E   tree
 D->A   back
 E->F   tree
 F->E   back
 G->C   cross
 G->F   cross
tree 5, back 2, forward 1, cross 3  (total 11)
the forest has 2 trees, so 7 - 2 = 5 tree edges
parenthesis structure: (A (B (D D) B) (C (E (F F) E) C) A) (G G)
the graph has 2 back edge(s), so it is NOT acyclic

=== Problem J  connected components ===
vertex     1   2   3   4   5   6   7   8   9  10  11  12
comp       0   0   0   1   1   1   1   2   2   2   2   2
number of components: 3
  1 and 3  connected? yes
  1 and 4  connected? no
  5 and 7  connected? yes
  6 and 11 connected? no
  8 and 12 connected? yes
edges 11, vertices 12, components 3; a forest would need 9 edges, this has 11

=== Problem K  bipartiteness ===
G1: BIPARTITE; sides {ADEFI} and {BCGH}
G1 plus the edge E-F: NOT bipartite; the edge E-F joins two vertices of the same color. Odd
cycle: EBACFE

=== Problem L  cycle detection ===
  path A-B-C-D          : acyclic (a forest)
  square A-B-C-D-A      : has a cycle
  G1                    : has a cycle
  two parallel edges A=B: has a cycle
  G2                     : has a directed cycle
  A->B, A->C, B->C       : is a DAG
  A->B, B->C, C->A       : has a directed cycle
```

Every number in Sections 12.4.2 to 12.4.7 is reproduced there, including the five-vertex odd cycle `EBACFE` and the parallel-edge case that a careless parent rule would miss.

12.5 Further Worked Problems III: DAGs and Topological Sort

12.5.1 The prerequisite DAG of this section

A degree program has eight subjects. An arrow $X \to Y$ means X *must be passed before* Y *may be taken*, and each subject takes the number of weeks shown in **Table 12.27**.

Table 12.27 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

Adjacency lists are alphabetical; the outer loop of any depth-first search runs $A, B, \ldots, H$; Kahn's queue is first-in-first-out and is initialized with the zero-in-degree vertices in alphabetical order.

12.5.2 Worked Problem M — is it a DAG?

Problem. (a) Run a depth-first search on the digraph of **Table 12.27** and give $d[v]/f[v]$ for all eight subjects. (b) Classify all nine arcs. (c) State, with the theorem you are using, whether the digraph is a DAG.

What it tests. The back-edge criterion of §11.2.3, and the discipline of finishing one subtree completely before starting the next.

Solution.

(a) The search. $\mathrm{Adj}[A] = [D]$, $\mathrm{Adj}[B] = [D, E]$, $\mathrm{Adj}[C] = [G]$, $\mathrm{Adj}[D] = [E, F]$, $\mathrm{Adj}[E] = [H]$, $\mathrm{Adj}[F] = [H]$, $\mathrm{Adj}[G] = [H]$, $\mathrm{Adj}[H] = [\,]$.

- A white: gray, $d[A] = 1$. $\to D$ white: gray, $d[D] = 2$.
- $\mathrm{Adj}[D]$, first E: white, gray, $d[E] = 3$. $\to H$ white: gray, $d[H] = 4$. H has no arcs: black, $f[H] = 5$. Back in E: exhausted, black, $f[E] = 6$.
- $\mathrm{Adj}[D]$, next F: white, gray, $d[F] = 7$. $\mathrm{Adj}[F] = [H]$: **black**, and $d[F] = 7 > 4 = d[H]$, so **cross edge**. F black, $f[F] = 8$.
- D exhausted: black, $f[D] = 9$. A exhausted: black, $f[A] = 10$.
- Outer loop: B white, gray, $d[B] = 11$. $\mathrm{Adj}[B] = [D, E]$: both **black** with smaller d, so both **cross**. B black, $f[B] = 12$.
- C white, gray, $d[C] = 13$. $\to G$ white, gray, $d[G] = 14$. $\mathrm{Adj}[G] = [H]$: black, smaller d, **cross**. G black, $f[G] = 15$. C black, $f[C] = 16$.
- D, E, F, G, H are all black; the loop ends. Clock $16 = 2|V|$. ✓

Table 12.28 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

(b) The classification. Using the timestamps of **Table 12.28**: tree $A \to D$, $D \to E$, $D \to F$, $E \to H$, $C \to G$ (five); cross $B \to D$, $B \to E$, $F \to H$, $G \to H$ (four); forward none; **back none**.

(c) The verdict. By the DAG theorem (Theorem 11.1, §11.2.3), *a directed graph is acyclic if and only if a depth-first search of it yields no back edges.* Zero back edges were found, so the digraph **is a DAG** — which had better be true, or the degree could never be completed.

THE TRAP. People frequently label $B \to D$ a *forward* edge because D "comes later". It is not: forward means the far endpoint is a **descendant of this vertex in the depth-first forest**, and D sits in A's tree, not B's. The timestamp test decides it mechanically: $d[B] = 11 > 2 = d[D]$, and a descendant always has the *larger* discovery time. Cross.

Answer. d/f as in **Table 12.28**; five tree and four cross arcs, no back arc; the digraph is a DAG.

CHECK YOURSELF. Would the digraph still be a DAG if the program added (a) the prerequisite $F \to E$? (b) the prerequisite $H \to D$?

Answer. (a) **Yes.** A cycle through the new arc would need a path from E back to F, and E's only way onward is $E \to H$, a dead end. (b) **No.** $D \to E \to H \to D$ is a cycle; a depth-first search would examine $H \to D$ while D is gray (inside D's own call, via E) and report a back edge.

12.5.3 Worked Problem N — topological sort by finishing time

Problem. (a) Produce a topological order of the prerequisite DAG by listing the vertices in decreasing order of finishing time, showing the finishing times. (b) Verify your answer by checking that every arc points forward. (c) Why does the algorithm use f and not d? Give a two-vertex example where sorting by d would fail.

What it tests. Algorithm 1 of §11.3.4 and the reason it is correct (Theorem 11.2).

Solution.

(a) The order. Sort by f descending: $C(16)$, $G(15)$, $B(12)$, $A(10)$, $D(9)$, $F(8)$, $E(6)$, $H(5)$. So

$$\mathrm{C} \prec \mathrm{G} \prec \mathrm{B} \prec \mathrm{A} \prec \mathrm{D} \prec \mathrm{F} \prec \mathrm{E} \prec \mathrm{H}.$$

In practice you do not sort at all: push each vertex onto the front of a list the instant it turns black, and the list *is* the answer when the search ends. That keeps the whole algorithm at $\Theta(V + E)$; an explicit sort would add $\Theta(V \lg V)$ for nothing.

(b) The verification. Positions: C 1, G 2, B 3, A 4, D 5, F 6, E 7, H 8. Every arc:

$A \to D$: $4 < 5$ ✓. $B \to D$: $3 < 5$ ✓. $B \to E$: $3 < 7$ ✓. $C \to G$: $1 < 2$ ✓. $D \to E$: $5 < 7$ ✓. $D \to F$: $5 < 6$ ✓. $E \to H$: $7 < 8$ ✓. $F \to H$: $6 < 8$ ✓. $G \to H$: $2 < 8$ ✓.

All nine point forward, so the order is legal. This check costs $\Theta(V + E)$ and belongs in every answer and every program.

(c) Why f and not d. Because *discovery* order says nothing about dependency. Take the two-vertex DAG with the single arc $B \to A$, and let the outer loop consider A first. Then $d[A] = 1$, $d[B] = 3$, and sorting by increasing d gives A, B — but the arc demands B before A, so the order is wrong. Sorting by decreasing f gives $f[A] = 2$, $f[B] = 4$, hence B, A — correct.

The reason, stated properly: when an arc $u \to v$ is examined, v is white (then v finishes inside u's call), or black (it has finished already), or gray — which is impossible here, since gray would mean a back edge and the graph is acyclic. In every case

$$f[v] < f[u] \qquad \text{for every arc } u \to v \text{ of a DAG,} \tag{12.15}$$

and so, by Equation (12.15), decreasing f always puts u before v.

The dressing problem of §11.3.3 is the classic instance, **Figure 12.16**, and its answer is **Figure 12.17** — the same nine vertices laid out in decreasing finishing time, with every arrow now pointing right.

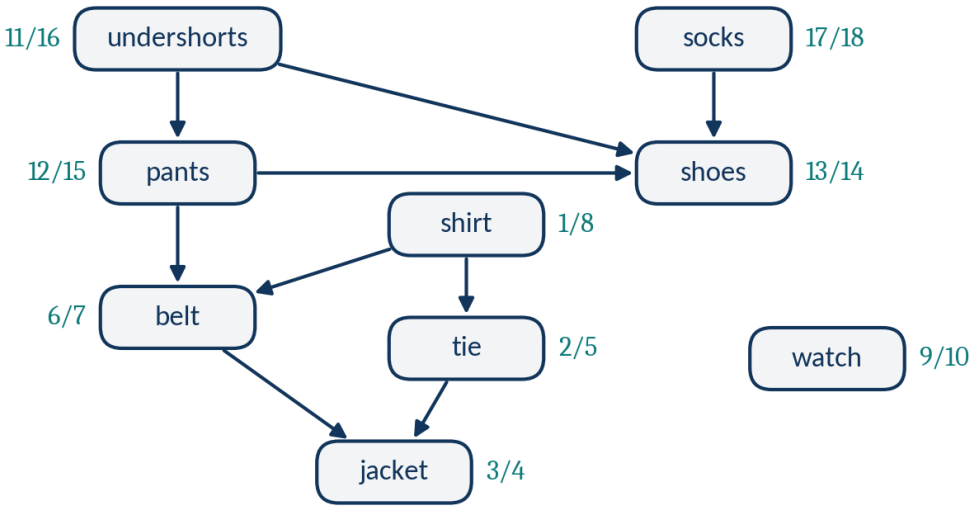


Figure 12.16 The precedence DAG for getting dressed, with the depth-first timestamps beside each vertex: shirt 1/8, tie 2/5, jacket 3/4, belt 6/7, watch 9/10, undershorts 11/16, pants 12/15, shoes 13/14, socks 17/18. Nine vertices, nine arcs, eighteen stamps.

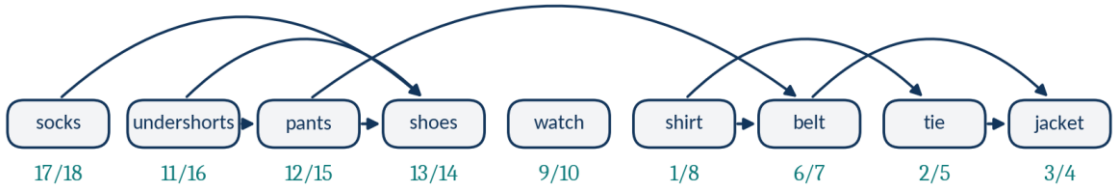


Figure 12.17 The result: the same nine garments laid out left to right in decreasing order of finishing time — socks 18, undershorts 16, pants 15, shoes 14, watch 10, shirt 8, belt 7, tie 5, jacket 4 — and now every arrow points right. That is what a topological sort *is*.

Answer. C, G, B, A, D, F, E, H; all nine arcs point forward; finishing times work because of Equation (12.15), and discovery times fail on $B \to A$ searched from A.

> **CHECK YOURSELF.** Run the depth-first method again with the outer loop in the order H, G, F, E, D, C, B, A. What order does it produce, and is it legal?
>
> *Answer.* Each outer call now starts at a vertex whose successors are all already finished, so every call finishes immediately: H 1/2, G 3/4, F 5/6, E 7/8, D 9/10, C 11/12, B 13/14, A 15/16. Decreasing f gives A, B, C, D, E, F, G, H — alphabetical, and **legal**: every one of the nine arcs goes from an earlier letter to a later one. A different order, equally correct.

12.5.4 Worked Problem O — topological sort by Kahn's algorithm

Problem. (a) Give the initial in-degree of every subject. (b) Run Kahn's algorithm, **Pseudocode 12.6**, giving a table with one row per round: the subject emitted, the in-degrees after

the removal, and the queue afterward. (c) Compare your answer with Problem N. On how many positions do the two orders agree, and why is that not a problem?

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
```

What it tests. Algorithm 2 of §11.3.9, executed exactly, and the understanding that a DAG usually has many topological orders.

Solution.

(a) In-degrees. Count the arrows arriving: A 0, B 0, C 0, D 2 (from A, B), E 2 (from B, D), F 1 (from D), G 1 (from C), H 3 (from E, F, G). Their sum is $0 + 0 + 0 + 2 + 2 + 1 + 1 + 3 = 9 = |E|$, which is the check to run before starting.

(b) The run. The queue starts with the zero-in-degree vertices in alphabetical order, $[A, B, C]$, and **Table 12.29** records every round.

Table 12.29 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)

Order: **A, B, C, D, G, E, F, H.**

(c) The comparison. The depth-first order was C, G, B, A, D, F, E, H; Kahn's is A, B, C, D, G, E, F, H. They agree on **one** position out of eight — the last, H — and disagree everywhere else.

That is not a problem. A topological order is any linear order in which every arc points forward, and the DAG constrains only *some* pairs. Nothing in the data relates A to C, so either may come first; nothing relates G to D; and so on. Both orders were verified arc by arc, and both pass. The only vertex forced into last place is H, because every other subject reaches it.

> **WHICH ALGORITHM TO USE, IN ONE LINE EACH.** Use **Kahn** when you want cycle detection for free, when you want to process vertices in waves (everything at in-degree zero can run in parallel), or when recursion depth is a worry. Use the **depth-first** method when you already

have a DFS in your program, or when you need the finishing times for something else as well. Both are $\Theta(V + E)$; nothing else separates them.

Answer. In-degrees 0,0,0,2,2,1,1,3; Kahn's order A, B, C, D, G, E, F, H; one position in common with Problem N, which is fine because both are legal.

CHECK YOURSELF. Replace Kahn's FIFO queue by a min-priority queue that always releases the alphabetically smallest ready subject. What order results?

Answer. Ready $\{A, B, C\}$: emit A (D drops to 1). Ready $\{B, C\}$: emit B (D to 0, E to 1). Ready $\{C, D\}$: emit C (G ready). Ready $\{D, G\}$: emit D (E, F ready). Ready $\{E, F, G\}$: emit E, then F, then G, and finally H. Order A, B, C, D, E, F, G, H — the alphabetically first of all legal orders, which is what a priority queue on names always produces.

12.5.5 Worked Problem P — counting the topological orders

Problem. (a) How many distinct topological orders does the prerequisite DAG have? Explain how to count them without listing all 8! permutations. (b) How many extra prerequisites must be added to make the order unique?

What it tests. The meaning of a topological order as a *linear extension* of a partial order, and the criterion for uniqueness.

Solution.

(a) The count. There are $8! = 40\,320$ permutations of eight subjects; the number that are legal topological orders is **84**. That figure comes from testing each permutation against the nine arcs, which is $40\,320 \times 9$ comparisons — fast enough for eight vertices and hopeless for twenty.

The method that does not enumerate is recursive. Let $N(S)$ be the number of legal orders of a set S of subjects; choose which subject comes first — it must be one with no prerequisite left in S — and count the orders of the rest:

$$N(\varnothing) = 1, \qquad N(S) = \sum_{\substack{v \in S \\ v \text{ has no prerequisite in } S}} N(S \setminus \{v\}). \tag{12.16}$$

Memoizing Equation (12.16) on the *set* S gives $O(2^n \cdot n)$ work — for $n = 8$ that is $256 \times 8 = 2048$ steps rather than 40 320, and the gap widens rapidly. Counting linear extensions is #P-complete in general [10], so no polynomial method is known; the point is the exponential-but-far-better idea, not a miracle.

A sanity check on 84: the three roots A, B, C can be interleaved fairly freely, H is forced last, and E and F are both after D but unrelated to each other. If every subject were independent the answer would be $8! = 40\,320$; if the DAG were a single chain it would be 1. Eighty-four is between the two and much nearer the chain, which is what nine constraints on eight vertices should do.

(b) Forcing uniqueness. A topological order is unique **if and only if the DAG contains a Hamiltonian path** — a directed path visiting every vertex — because such a path forces every pair into a fixed relative order, and without one some adjacent pair of the order is unrelated and could be swapped. In Kahn's terms: the order is unique exactly when the queue never holds more than one vertex.

So we need to add arcs until some legal order has *all seven* of its neighboring pairs joined by an arc. Look first at why one arc cannot be enough: A, B and C are all sources, and a single new arc can take away the source status of only one of them, so two sources survive and either may come

first. The exact minimum comes from checking every one of the 84 legal orders and counting how many of its seven neighboring pairs are *already* arcs. The best is 4, achieved by

$$A,\ B,\ D,\ E,\ C,\ G,\ F,\ H,$$

whose neighboring pairs $B \to D$, $D \to E$, $C \to G$ and $F \to H$ are existing prerequisites. The other three — $A \to B$, $E \to C$ and $G \to F$ — must be added, and adding them creates no cycle. So **three extra prerequisites are necessary and sufficient**: no order has more than four of its links already present, so no two additions can complete a Hamiltonian path. **Listing 12.8** performs exactly this check.

Answer. 84 legal orders, countable by Equation (12.16); three extra prerequisites (for instance $A \to B, E \to C, G \to F$) are needed to make the order unique.

> **CHECK YOURSELF.** Add the single prerequisite $F \to E$. How many legal orders remain?
>
> *Answer.* **42**, exactly half. Before the change, E and F were interchangeable: both need D (and E also needs B, which already precedes D), and both are needed only by H. So swapping E and F turns any legal order into another legal one, pairing the 84 orders off. The new arc keeps exactly the half in which F precedes E.

12.5.6 Worked Problem Q — the critical path

Problem. (a) Compute the earliest week each subject can be finished, assuming unlimited teaching capacity, and give the critical path. (b) How long would the program take if the subjects were taken strictly one at a time? (c) Add the prerequisite $H \to A$. What now happens to each of the two topological-sort algorithms?

What it tests. Longest paths in a DAG by one relaxation per arc in topological order — Equation (12.2) of §12.2.5 again, with weeks instead of minutes.

Solution.

(a) Earliest finishing weeks. Process the subjects in *any* topological order — Kahn's will do — and apply Equation (12.2): a source finishes at its own duration; otherwise $ef[v] = \max_{u \to v} ef[u] + weeks[v]$.

- $ef[A] = 3$. $ef[B] = 2$. $ef[C] = 4$.
- D: predecessors $A(3)$, $B(2)$; max $= 3$; $ef[D] = 3 + 5 = 8$.
- G: predecessor $C(4)$; $ef[G] = 4 + 2 = 6$.
- E: predecessors $B(2)$, $D(8)$; max $= 8$; $ef[E] = 8 + 6 = 14$.
- F: predecessor $D(8)$; $ef[F] = 8 + 3 = 11$.
- H: predecessors $E(14)$, $F(11)$, $G(6)$; max $= 14$; $ef[H] = 14 + 4 = \mathbf{18}$.

Tracing the maxima backward, H took its time from E, E from D, D from A:

$$\mathrm{A} \to \mathrm{D} \to \mathrm{E} \to \mathrm{H}, \qquad 3 + 5 + 6 + 4 = 18 \text{ weeks.}$$

That is the **critical path**: lengthen any subject on it and the whole degree lengthens; shorten any subject off it and nothing changes. G, for instance, finishes in week 6 but is not needed until week 14, so it carries $14 - 6 = 8$ weeks of slack.

(b) One at a time. $3 + 2 + 4 + 5 + 6 + 3 + 2 + 4 = 29$ weeks. The DAG saves $29 - 18 = 11$ weeks, and that gap is exactly the amount of parallel study the precedences permit.

(c) Adding $H \to A$. The graph now contains the directed cycle $A \to D \to E \to H \to A$, and both algorithms report it, in their own idioms.

- **Kahn** emits only **3** of the 8 subjects. In-degree never reaches zero for the five vertices on or after the cycle, so the queue empties early. The five it failed to emit name the problem.
- **The depth-first method** finds **1** back edge, $H \to A$, examined while A is still gray. By the DAG theorem that single back edge is the certificate.

Neither produces a "nearly right" order, which is the important property: a broken specification must fail loudly. A topological sort that silently returned a partial list would be worse than useless in a scheduler.

Answer. Earliest finishes A 3, B 2, C 4, D 8, G 6, E 14, F 11, H 18; critical path $A \to D \to E \to H$, 18 weeks against 29 one at a time; with $H \to A$ both algorithms report a cycle.

> **CHECK YOURSELF.** Algorithms (E) is shortened from 6 weeks to 3. What is the new finishing week, and what is the critical path?
>
> *Answer.* $ef[E] = 8 + 3 = 11$, and $ef[F] = 11$ as before, so $ef[H] = \max(11,11,6) + 4 = 15$. The program shrinks to **15 weeks**, and there are now **two** critical paths, $A \to D \to E \to H$ and $A \to D \to F \to H$, both of 15 weeks. Shortening E by three weeks saved only three weeks, and any further shortening of E alone would save nothing, because F is now just as critical.

12.5.7 Problems M to Q as one program

Listing 12.8 answers Problems M to Q mechanically, including the count of legal orders, the fewest extra prerequisites that force a unique order, and the injected cycle of Problem Q.

Listing 12.8 Worked Problems M to Q: the depth-first timestamps and classification, both topological sorts, the count of legal orders and the uniqueness test, the critical path, and the injected cycle. Compile with `g++ -std=c++17 -O2 -Wall -o dag dag.cpp`.

```
// Worked Problems M to Q of Chapter 12 -- DAGs and topological sort.
//   M  is the prerequisite digraph a DAG?  (DFS, and the back-edge test)
//   N  topological sort by depth-first finishing times
//   O  topological sort by Kahn's in-degree algorithm
//   P  how many distinct topological orders are there, and how many extra
//      prerequisites would force a unique one?
//   Q  the critical path: the fastest the whole curriculum can be finished
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <numeric>
#include <functional>
using namespace std;

static const vector<string> NAME = {
    "A ProgBasics", "B DiscreteMath", "C Calculus", "D DataStruct",
    "E Algorithms", "F Databases",   "G Statistics", "H MachineLearn"
};
static const vector<int> WEEKS = { 3, 2, 4, 5, 6, 3, 2, 4 };
static const vector<pair<int,int>> PRE = {
    {0,3},{1,3},{1,4},{2,6},{3,4},{3,5},{4,7},{5,7},{6,7}
};
static const int N = 8;
static string tag(int v) { return NAME[v].substr(0, 1); }

vector<vector<int>> build(const vector<pair<int,int>>& E) {
    vector<vector<int>> adj(N);
    for (auto [u,v] : E) adj[u].push_back(v);
    for (auto& l : adj) sort(l.begin(), l.end());
    return adj;
}

int main() {
    auto adj = build(PRE);
```

Listing 12.8 (Continued) *Worked Problems M to Q: the depth-first timestamps and classification, both topological sorts, the count of legal orders and the uniqueness test, the critical path, and the injected cycle. Compile with g++ -std=c++17 -O2 -Wall -o dag dag.cpp.*

```
    cout << "=== Problem M  depth-first search, and the back-edge test ===\n";
    vector<int> col(N,0), d(N,0), f(N,0), pi(N,-1);
    vector<string> cls(PRE.size());
    int clk = 0;
    function<void(int)> visit = [&](int u) {
        col[u] = 1; d[u] = ++clk;
        for (int v : adj[u]) {
            int idx = -1;
            for (size_t i = 0; i < PRE.size(); ++i)
                if (PRE[i].first == u && PRE[i].second == v) idx = (int)i;
            if (col[v] == 0)      { cls[idx] = "tree";    pi[v] = u; visit(v); }
            else if (col[v] == 1) { cls[idx] = "BACK"; }
            else                  { cls[idx] = (d[u] < d[v] ? "forward" : "cross"); }
        }
        col[u] = 2; f[u] = ++clk;
    };
    for (int v = 0; v < N; ++v) if (col[v] == 0) visit(v);
    cout << "v      "; for (int v = 0; v < N; ++v) cout << setw(7) << tag(v); cout << "\n";
    cout << "d/f    "; for (int v = 0; v < N; ++v) cout << setw(7) << (to_string(d[v]) +
"/" + to_string(f[v])); cout << "\n";
    cout << "clock stopped at " << clk << " = 2|V| = " << 2*N << "\n";
    int back = 0;
    for (size_t i = 0; i < PRE.size(); ++i) {
        cout << "  " << tag(PRE[i].first) << "->" << tag(PRE[i].second) << "  " << cls[i]
<< "\n";
        if (cls[i] == "BACK") ++back;
    }
    cout << "back edges: " << back << "  -> the digraph IS "
         << (back ? "NOT a DAG" : "a DAG") << "\n";

    cout << "\n=== Problem N  topological sort by decreasing finishing time ===\n";
    vector<int> byF(N); iota(byF.begin(), byF.end(), 0);
    sort(byF.begin(), byF.end(), [&](int a, int b){ return f[a] > f[b]; });
    cout << "  order:";
    for (int v : byF) cout << " " << tag(v) << "(" << f[v] << ")";
    cout << "\n";
    bool ok = true; vector<int> pos(N);
    for (int i = 0; i < N; ++i) pos[byF[i]] = i;
    for (auto [u,v] : PRE) if (pos[u] >= pos[v]) ok = false;
    cout << "  every arc points forward? " << (ok ? "yes" : "NO") << "\n";
```

Listing 12.8 (Continued) *Worked Problems M to Q: the depth-first timestamps and classification, both topological sorts, the count of legal orders and the uniqueness test, the critical path, and the injected cycle. Compile with g++ -std=c++17 -O2 -Wall -o dag dag.cpp.*

```
    cout << "\n=== Problem O  topological sort by Kahn's in-degree algorithm ===\n";
    vector<int> indeg(N, 0);
    for (auto [u,v] : PRE) ++indeg[v];
    cout << "  initial in-degrees: ";
    for (int v = 0; v < N; ++v) cout << tag(v) << "=" << indeg[v] << " ";
    cout << "\n";
    queue<int> Q;
    for (int v = 0; v < N; ++v) if (indeg[v] == 0) Q.push(v);
    vector<int> kahn;
    cout << "  round  emit  in-degrees after         queue after\n";
    int round = 0;
    while (!Q.empty()) {
        int u = Q.front(); Q.pop(); kahn.push_back(u);
        for (int v : adj[u]) if (--indeg[v] == 0) Q.push(v);
        string s; for (int v = 0; v < N; ++v) s += to_string(indeg[v]) + " ";
        queue<int> t = Q; string qs;
        while (!t.empty()) { qs += tag(t.front()) + " "; t.pop(); }
        cout << setw(7) << ++round << setw(6) << tag(u) << "  " << left << setw(23) << s
             << right << (qs.empty() ? "(empty)" : qs) << "\n";
    }
    cout << "  emitted " << kahn.size() << " of " << N << " -> "
         << (kahn.size() == (size_t)N ? "acyclic" : "a cycle exists") << "\n";
    cout << "  order:";
    for (int v : kahn) cout << " " << tag(v);
    cout << "\n  DFS order:";
    for (int v : byF) cout << " " << tag(v);
    cout << "\n";
    int agree = 0; for (int i = 0; i < N; ++i) if (byF[i] == kahn[i]) ++agree;
    cout << "  positions on which the two orders agree: " << agree << " of " << N << "\n";

    cout << "\n=== Problem P  how many distinct topological orders? ===\n";
    vector<int> perm(N); iota(perm.begin(), perm.end(), 0);
    long long total = 0, legal = 0;
    int bestChain = -1; vector<int> bestOrder;
    do {
        ++total;
        vector<int> p(N); for (int i = 0; i < N; ++i) p[perm[i]] = i;
        bool good = true;
        for (auto [u,v] : PRE) if (p[u] >= p[v]) { good = false; break; }
        if (!good) continue;
        ++legal;
        // how many neighboring pairs of this order are already prerequisites?
        int chain = 0;
        for (int i = 0; i + 1 < N; ++i)
            for (auto [u,v] : PRE) if (u == perm[i] && v == perm[i+1]) ++chain;
        if (chain > bestChain) { bestChain = chain; bestOrder = perm; }
```

Listing 12.8 (Continued) *Worked Problems M to Q: the depth-first timestamps and classification, both topological sorts, the count of legal orders and the uniqueness test, the critical path, and the injected cycle. Compile with g++ -std=c++17 -O2 -Wall -o dag dag.cpp.*

```
    } while (next_permutation(perm.begin(), perm.end()));
    cout << "  permutations checked: " << total << "\n";
    cout << "  legal topological orders: " << legal << "\n";
    // a unique order needs a Hamiltonian path; every missing link must be added
    cout << "  best legal order for a Hamiltonian path:";
    for (int v : bestOrder) cout << " " << tag(v);
    cout << "\n  it already contains " << bestChain << " of its " << N - 1
         << " links, so at least " << N - 1 - bestChain
         << " extra prerequisites are needed to force a unique order\n";

    cout << "\n=== Problem Q  the critical path ===\n";
    vector<int> ef(N, 0), pred(N, -1);
    for (int u : kahn) {
        if (ef[u] == 0) ef[u] = WEEKS[u];
        for (int v : adj[u]) if (ef[u] + WEEKS[v] > ef[v]) { ef[v] = ef[u] + WEEKS[v];
pred[v] = u; }
    }
    cout << "  subject          weeks  earliest finish\n";
    for (int v : kahn)
        cout << "  " << left << setw(16) << NAME[v] << right << setw(5) << WEEKS[v]
             << setw(16) << ef[v] << "\n";
    int last = (int)(max_element(ef.begin(), ef.end()) - ef.begin());
    vector<int> path;
    for (int x = last; x != -1; x = pred[x]) path.push_back(x);
    reverse(path.begin(), path.end());
    cout << "  critical path (" << ef[last] << " weeks):";
    for (size_t i = 0; i < path.size(); ++i) cout << (i ? " -> " : " ") << tag(path[i]);
    cout << "\n  total of all eight subjects taken one at a time: "
         << accumulate(WEEKS.begin(), WEEKS.end(), 0) << " weeks\n";

    cout << "\n=== the trap: one extra prerequisite H -> A ===\n";
    vector<pair<int,int>> bad = PRE; bad.push_back({7,0});
    auto adj2 = build(bad);
    vector<int> id2(N, 0);
    for (auto [u,v] : bad) ++id2[v];
    queue<int> Q2; for (int v = 0; v < N; ++v) if (id2[v] == 0) Q2.push(v);
    int emitted = 0;
    while (!Q2.empty()) { int u = Q2.front(); Q2.pop(); ++emitted;
        for (int v : adj2[u]) if (--id2[v] == 0) Q2.push(v); }
    cout << "  Kahn emits " << emitted << " of " << N
         << " -> the remaining " << N - emitted << " vertices lie on or after a cycle\n";
    vector<int> c2(N, 0); int nb = 0;
    function<void(int)> v2 = [&](int u) {
        c2[u] = 1;
        for (int v : adj2[u]) { if (c2[v] == 1) ++nb; else if (c2[v] == 0) v2(v); }
        c2[u] = 2;
    };
```

Listing 12.8 (Continued) *Worked Problems M to Q: the depth-first timestamps and classification, both topological sorts, the count of legal orders and the uniqueness test, the critical path, and the injected cycle. Compile with g++ -std=c++17 -O2 -Wall -o dag dag.cpp.*

```
    for (int v = 0; v < N; ++v) if (!c2[v]) v2(v);
    cout << "  DFS finds " << nb << " back edge(s) -> the same verdict, by a different
route\n";
    return 0;
}
```

The complete output confirms every hand-computed value:

```
=== Problem M  depth-first search, and the back-edge test ===
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
clock stopped at 16 = 2|V| = 16
  A->D  tree
  B->D  cross
  B->E  cross
  C->G  tree
  D->E  tree
  D->F  tree
  E->H  tree
  F->H  cross
  G->H  cross
back edges: 0  -> the digraph IS a DAG

=== Problem N  topological sort by decreasing finishing time ===
  order: C(16) G(15) B(12) A(10) D(9) F(8) E(6) H(5)
  every arc points forward? yes

=== Problem O  topological sort by Kahn's in-degree algorithm ===
  initial in-degrees: A=0 B=0 C=0 D=2 E=2 F=1 G=1 H=3
  round  emit  in-degrees after         queue after
      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  0 0 0 0 0 0 0 0        (empty)
  emitted 8 of 8 -> acyclic
  order: A B C D G E F H
  DFS order: C G B A D F E H
  positions on which the two orders agree: 1 of 8

=== Problem P  how many distinct topological orders? ===
  permutations checked: 40320
  legal topological orders: 84
  best legal order for a Hamiltonian path: A B D E C G F H
  it already contains 4 of its 7 links, so at least 3 extra prerequisites are needed to
force a unique order

=== Problem Q  the critical path ===
  subject          weeks  earliest finish
  A ProgBasics        3               3
  B DiscreteMath      2               2
  C Calculus          4               4
  D DataStruct        5               8
  G Statistics        2               6
  E Algorithms        6              14
  F Databases         3              11
  H MachineLearn      4              18
  critical path (18 weeks): A -> D -> E -> H
  total of all eight subjects taken one at a time: 29 weeks

=== the trap: one extra prerequisite H -> A ===
  Kahn emits 3 of 8 -> the remaining 5 vertices lie on or after a cycle
  DFS finds 1 back edge(s) -> the same verdict, by a different route
```

12.6 Further Worked Problems IV: Minimum Spanning Trees

12.6.1 The weighted graph of this section

Problems R, S, U, V and W use one graph. G is undirected, connected and weighted, on $V = \{A, \dots, H\}$ with the thirteen edges of **Table 12.30**.

Table 12.30 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

Break every tie by the alphabetical order of the endpoints. Drawn out, G is a chain of triangles — ABC, BCD (through BD), CDE, DEF, EFG, FGH — which is why so many of its cycles are short. Problem T uses the nine-vertex graph of **Figure 12.1** instead.

12.6.2 Worked Problem R — Kruskal, with a disjoint-set trace

Problem. Run **Kruskal's algorithm** on G. Give a table with one row per edge examined, in the order examined, showing the representative returned by `Find` for each endpoint, the accept or reject decision, and the disjoint-set forest (`parent` and `rank` arrays) after the step. State the tree and its weight, and say how you know when to stop.

What it tests. That Kruskal is really *two* algorithms interleaved — a sort, and a disjoint-set structure (§11.10) — and that you can write down what the second one is doing.

Solution.

The sorted edge list. Weight first, then alphabetical:

$$BC\ 1,\ DE\ 2,\ AB\ 3,\ FG\ 3,\ CD\ 4,\ AC\ 5,\ BD\ 6,\ EF\ 7,\ DF\ 8,\ CE\ 9,\ EG\ 10,\ GH\ 1$$

The run. `parent` is written as eight letters, one per vertex A to H; `rank` as eight digits. The implementation is the one of **Listing 12.1**: `Union` attaches the root of lower rank beneath the root of higher rank, and on a tie attaches the second root beneath the first and raises the first root's rank; `Find` compresses the path it walks.

Table 12.31 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
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

Follow three rows in detail, because they show every behavior of the structure. **Row 3**, AB: `Find(A)` $=A$ (rank 0) and `Find(B)` $=B$ (rank 1). The ranks differ, so the lower-ranked root A goes under B, and no rank changes. **Row 5**, CD: `Find(C)` $=B$ (rank 1) and `Find(D)` $=D$ (rank 1). A tie: D goes under B and `rank[B]` rises to 2. **Row 8**, EF: `Find(E)` walks $E \to D \to B$ and compresses, setting `parent[E]` to B; `Find(F)` $=F$ (rank 1). B has rank 2, so F goes under B.

The stopping rule. After row 12 the tree has $7 = |V| - 1$ edges, and a spanning tree on eight vertices cannot have more. The thirteenth edge, FH 12, need never be examined: whatever it is, both its endpoints are already joined. Stopping at $|V| - 1$ saves one comparison here and can save a great many on a dense graph.

The answer.

$$T = \{BC\ 1,\ DE\ 2,\ AB\ 3,\ FG\ 3,\ CD\ 4,\ EF\ 7,\ GH\ 11\}, \qquad w(T) = 1 + 2 + 3 + 3 + 4 + 7 + 11 = \mathbf{31}.$$

> **THE TRAP IN THE RANKS.** `rank` is an *upper bound on height*, not the height. It rises only when two roots of **equal** rank are united (rows 1 to 5 show both behaviors), and path compression never lowers it. After the run, `rank[B]` is 2 while B's tree is only one level deep. That is harmless and deliberate: recomputing exact heights would cost more than it saves.

Figure 12.18 shows the step, on the nine-vertex graph of Chapter 11, at which Kruskal's answer parts company with Prim's — two equal-weight edges, opposite fates, decided entirely by the partition and not by the weight.

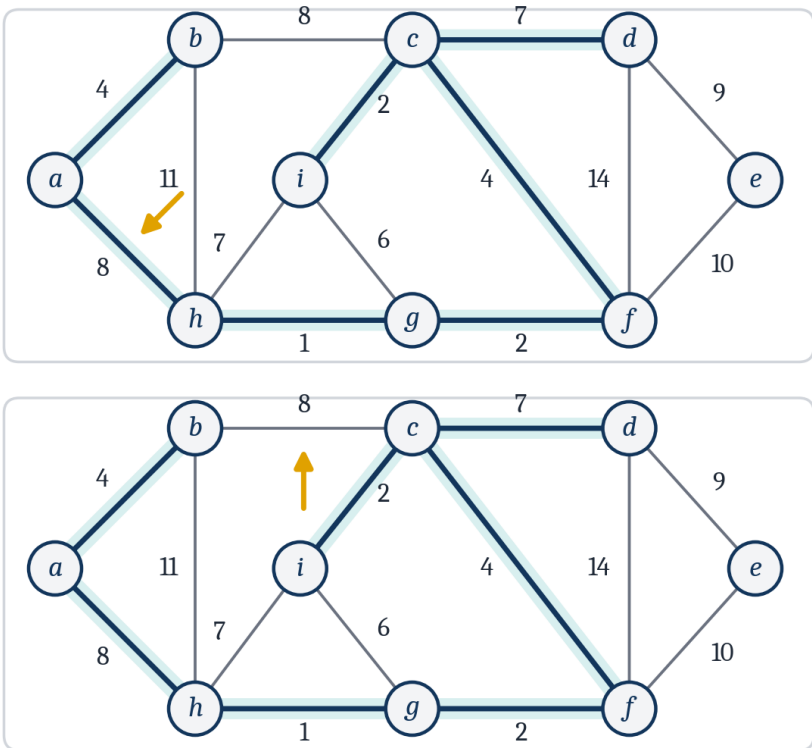


Figure 12.18 Kruskal on the nine-vertex graph of Chapter 11 at weight 8: (a, h) accepted, (b, c) rejected. Prim, on the same graph, does the opposite — it takes (b, c) and never considers (a, h). Both trees weigh 37. Equal weights are where the two algorithms are free to disagree.

Answer. $T = \{BC, DE, AB, FG, CD, EF, GH\}$, weight 31, found after examining twelve of the thirteen edges.

> **CHECK YOURSELF.** Raise $w(FG)$ from 3 to 13 and rerun Kruskal. Which edges does the tree contain now, and what does it weigh?
>
> *Answer.* BC 1, DE 2, AB 3, CD 4 are accepted as before; AC 5 and BD 6 are rejected; EF 7 is accepted; DF 8 and CE 9 are rejected; EG 10 is now **accepted**, because G is no longer joined to anything; GH 11 is accepted. The tree is $\{BC, DE, AB, CD, EF, EG, GH\}$, weight $1 + 2 + 3 + 4 + 7 + 10 + 11 = 38$.

12.6.3 Worked Problem S — Prim, with the key array

Problem. Run **Prim–Jarník** on G from A. Give a table with one row per extraction showing the vertex extracted, the edge committed, the running total, and the remaining vertices afterward as `v:key/π`. Compare the resulting tree with Problem R's.

What it tests. The two arrays of §11.9.2. For a vertex v not yet in the tree T,

$$key[v] = \min\{\, w(u, v) : u \in T,\ \{u, v\} \in E \,\}, \tag{12.17}$$

the weight of the cheapest edge found so far from the growing tree to v (∞ if there is none), and $\pi[v]$ is the tree end of that edge. Everything else is bookkeeping: extract the vertex of least key, commit its edge, and restore Equation (12.17) by lowering the keys of its neighbors.

Solution. Start with $key[A] = 0$, all others ∞, all eight vertices waiting.

Table 12.32 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)

The comparison. Reading **Table 12.32**, Prim's tree is $\{AB, BC, CD, DE, EF, FG, GH\}$, weight 31 — the *same set of edges* as Kruskal's, merely committed in a different order. Kruskal took them in weight order 1,2,3,3,4,7,11; Prim took them in the order the growing tree reached them, 3,1,4,2,7,3,11. Agreement on the edge set is not guaranteed in general (see **Figure 12.18**); agreement on the *weight* always is.

The Prim run here has an unusual shape: the tree grows as a path $A \to B \to C \to \cdots \to H$, so every extraction commits an edge to the vertex just added. That is a coincidence of this chain-of-triangles graph, not a property of Prim. On the nine-vertex graph of Chapter 11 the behavior is quite different, as Problem T shows — its cheapest edge, of weight 1, is not taken until the seventh step.

Answer. Prim from A commits $AB, BC, CD, DE, EF, FG, GH$ in that order, weight 31 — the same tree as Kruskal's.

> **CHECK YOURSELF.** Run Prim on G from H. In what order are the vertices extracted, and which edge does each commit?
>
> *Answer.* H (root), G by $GH(11)$, F by $FG(3)$, E by $EF(7)$, D by $DE(2)$, C by $CD(4)$, B by $BC(1)$, A by $AB(3)$. Total 31, the same seven edges, grown from the other end of the chain.

12.6.4 Worked Problem T — Prim, with a binary heap

Problem. The array of Problem S finds the minimum key by scanning every waiting vertex, which costs $\Theta(V)$ per extraction and $\Theta(V^2)$ in all. Real implementations use a binary heap. The C++ `std::priority_queue` has no `DecreaseKey`, so the standard trick (§11.9.10) is **lazy**: when a key falls, push a *new* entry and leave the old one in the heap; when an entry reaches the top, discard it if its vertex is already in the tree or its key is out of date. (a) Run this version, **Pseudocode 12.7**, on the nine-vertex graph of **Figure 12.1** from vertex a, with neighbors scanned alphabetically, and give the heap after every step. (b) Count the pushes, the pops and the stale pops. (c) Give the running time. (d) Compare the tree with the one of Chapter 11.

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
```

What it tests. The priority-queue implementation of Prim (§11.9.4 and §11.9.7), and exactly what a heap holds at each moment.

Solution.

(a) The run. The graph's fourteen edges are ab 4, ah 8, bc 8, bh 11, cd 7, cf 4, ci 2, de 9, df 14, ef 10, fg 2, gh 1, gi 6, hi 7. The heap is shown as its entries in increasing order, each written `vertex:key`; when two entries have the same key, the one with the alphabetically earlier vertex comes out first. **Table 12.33** is the run.

Table 12.33 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
9	e: 9	de(9)	37	—	e: 10
—	e: 10	stale, discarded	37	—	(empty)

Three moments deserve a closer look. **Step 2 to step 3:** c and h both have key 8, and c comes out first by the tie rule — exactly the choice made in §11.9.5. **Step 4:** h's key falls from 8 to 7, and the heap now holds *two* entries for h, the live h: 7 and the stale h: 8. **Step 6:** h's key falls again, to 1, and a third entry is pushed. When the stale copies h: 7 and h: 8 finally reach the top after step 8, the test on line 5 of **Pseudocode 12.7** sees that h is already in the tree and throws them away.

(b) The counts. Every push corresponds to a key being lowered, plus the initial push of the root: $1 + 12 = 13$ pushes. Every push is eventually popped: 13 pops, of which 9 are real extractions (one per vertex) and 4 are stale. **Listing 12.9** prints exactly these counts.

(c) The running time. Each edge can lower a key at most once: it is examined from both ends, but by the time its second end is extracted the first is already in the tree, and line 9 ignores it. So there are at most $|E| + 1$ pushes in all (here $12 + 1 = 13$ out of a possible $14 + 1$) and the heap never holds more than $|E| + 1$ entries. Each push and each pop costs $O(\lg E)$, and $\lg E \leq \lg V^2 = 2\lg V$, so the whole run costs

$$O(E \lg E) = O(E \lg V), \tag{12.18}$$

the same bound as Prim with a `DecreaseKey` heap, by Equation (12.18). The price of laziness is memory — the heap may hold up to $|E|$ entries instead of $|V|$ — and that price is usually worth paying for code this simple.

(d) The tree. The committed edges are ab 4, bc 8, ci 2, cf 4, fg 2, gh 1, cd 7, de 9, total 37 — the tree of **Figure 12.19**, edge for edge.

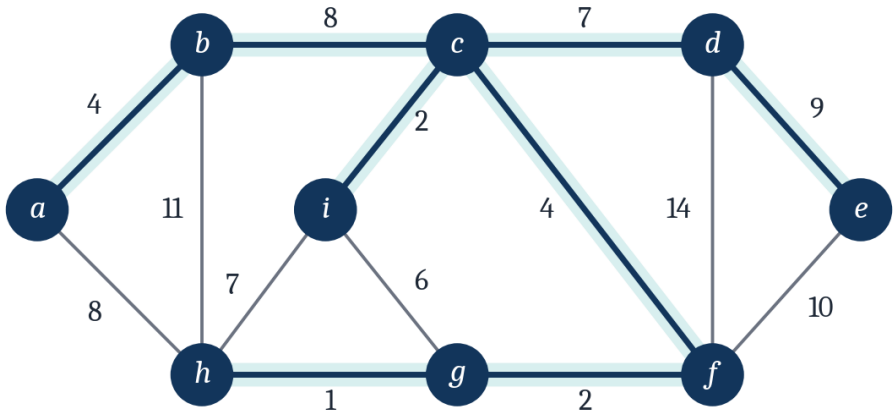


Figure 12.19 The finished minimum spanning tree on the nine-vertex graph: (a, b), (b, c), (c, i), (c, f), (f, g), (g, h), (c, d), (d, e), of total weight 37. Eight edges for nine vertices. Kruskal reaches a different set of eight edges — also weighing 37 — because two edges of weight 8 tie.

Listing 12.9 runs the lazy version twice, from a and from i, printing the heap after every step.

Listing 12.9 Prim's algorithm with `std::priority_queue` and lazy `DecreaseKey`, printing the heap after every step and counting the stale entries; run from a and from i. Compile with `g++ -std=c++17 -O2 -Wall -o primheap primheap.cpp`.

```
// Worked Problem T: Prim's algorithm with a binary-heap priority queue,
// traced on the nine-vertex graph of Chapter 11 (the graph of the revision
// figure). The heap is std::priority_queue with "lazy" DecreaseKey: an
// improved key is pushed as a new entry, and an entry whose key no longer
// matches key[v] (or whose vertex is already in the tree) is discarded when
// it reaches the top. The heap's contents are printed after every step.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
using namespace std;

static const string NM = "abcdefghi";
struct Edge { int a, b, w; };
static const vector<Edge> G = {
    {0,1,4}, {0,7,8}, {1,2,8}, {1,7,11}, {2,3,7}, {2,5,4}, {2,8,2},
    {3,4,9}, {3,5,14}, {4,5,10}, {5,6,2}, {6,7,1}, {6,8,6}, {7,8,7}
};
using Entry = pair<int,int>;                                    // (key, vertex)

string heapContents(priority_queue<Entry, vector<Entry>, greater<>> h) {
    string s;                                                   // pops a copy, smallest
first
    while (!h.empty()) { s += string(1, NM[h.top().second]) + ":" +
to_string(h.top().first) + " "; h.pop(); }
    return s.empty() ? "(empty)" : s;
}

void prim(int root) {
    int n = (int)NM.size();
    vector<vector<pair<int,int>>> adj(n);
    for (const Edge& e : G) { adj[e.a].push_back({e.b, e.w}); adj[e.b].push_back({e.a,
e.w}); }
    for (auto& l : adj) sort(l.begin(), l.end());               // neighbors alphabetically
    vector<int> key(n, INT_MAX), pi(n, -1);
    vector<char> inTree(n, 0);
    priority_queue<Entry, vector<Entry>, greater<>> Q;
    key[root] = 0; Q.push({0, root});
    int total = 0, pops = 0, stale = 0, pushes = 1, step = 0;
    cout << "=== Prim from " << NM[root] << " with a binary heap ===\n";
    while (!Q.empty()) {
        auto [k, u] = Q.top(); Q.pop(); ++pops;
        if (inTree[u] || k != key[u]) {                         // an out-of-date entry
            ++stale;
            cout << "      pop " << NM[u] << ":" << k << "  stale, discarded\n";
            continue;
```

Listing 12.9 (Continued) *Prim's algorithm with std::priority_queue and lazy DecreaseKey, printing the heap after every step and counting the stale entries; run from a and from i. Compile with g++ -std=c++17 -O2 -Wall -o primheap primheap.cpp.*

```
        }
        inTree[u] = 1; total += k;
        string edge = pi[u] < 0 ? "(root)" : string(1, NM[pi[u]]) + NM[u] + "(" +
to_string(k) + ")";
        string upd;
        for (auto [v, w] : adj[u])
            if (!inTree[v] && w < key[v]) {
                upd += string(1, NM[v]) + ":" + (key[v] == INT_MAX ? "inf" :
to_string(key[v]))
                        + "->" + to_string(w) + " ";
                key[v] = w; pi[v] = u; Q.push({w, v}); ++pushes;
            }
        cout << "step " << ++step << "  pop " << NM[u] << ":" << k << "  add " << left <<
setw(7) << edge
             << right << " total " << setw(2) << total << "  updates " << (upd.empty() ? "-
" : upd) << "\n"
             << "        heap: " << heapContents(Q) << "\n";
    }
    cout << "tree:";
    for (int v = 0; v < n; ++v) if (pi[v] >= 0) cout << " " << NM[pi[v]] << NM[v] << "(" <<
key[v] << ")";
    cout << "\ntotal weight " << total << "; heap pushes " << pushes << ", pops " << pops
         << ", stale pops " << stale << "\n";
}

int main() { prim(0); cout << "\n"; prim(8); return 0; }
```

```
=== Prim from a with a binary heap ===
step 1  pop a:0  add (root)  total  0  updates b:inf->4 h:inf->8
        heap: b:4 h:8
step 2  pop b:4  add ab(4)   total  4  updates c:inf->8
        heap: c:8 h:8
step 3  pop c:8  add bc(8)   total 12  updates d:inf->7 f:inf->4 i:inf->2
        heap: i:2 f:4 d:7 h:8
step 4  pop i:2  add ci(2)   total 14  updates g:inf->6 h:8->7
        heap: f:4 g:6 d:7 h:7 h:8
step 5  pop f:4  add cf(4)   total 18  updates e:inf->10 g:6->2
        heap: g:2 g:6 d:7 h:7 h:8 e:10
step 6  pop g:2  add fg(2)   total 20  updates h:7->1
        heap: h:1 g:6 d:7 h:7 h:8 e:10
step 7  pop h:1  add gh(1)   total 21  updates -
        heap: g:6 d:7 h:7 h:8 e:10
      pop g:6  stale, discarded
step 8  pop d:7  add cd(7)   total 28  updates e:10->9
        heap: h:7 h:8 e:9 e:10
      pop h:7  stale, discarded
      pop h:8  stale, discarded
step 9  pop e:9  add de(9)   total 37  updates -
        heap: e:10
      pop e:10  stale, discarded
tree: ab(4) bc(8) cd(7) de(9) cf(4) fg(2) gh(1) ci(2)
total weight 37; heap pushes 13, pops 13, stale pops 4

=== Prim from i with a binary heap ===
step 1  pop i:0  add (root)  total  0  updates c:inf->2 g:inf->6 h:inf->7
        heap: c:2 g:6 h:7
step 2  pop c:2  add ic(2)   total  2  updates b:inf->8 d:inf->7 f:inf->4
        heap: f:4 g:6 d:7 h:7 b:8
step 3  pop f:4  add cf(4)   total  6  updates e:inf->10 g:6->2
        heap: g:2 g:6 d:7 h:7 b:8 e:10
step 4  pop g:2  add fg(2)   total  8  updates h:7->1
        heap: h:1 g:6 d:7 h:7 b:8 e:10
step 5  pop h:1  add gh(1)   total  9  updates a:inf->8
        heap: g:6 d:7 h:7 a:8 b:8 e:10
      pop g:6  stale, discarded
step 6  pop d:7  add cd(7)   total 16  updates e:10->9
        heap: h:7 a:8 b:8 e:9 e:10
      pop h:7  stale, discarded
step 7  pop a:8  add ha(8)   total 24  updates b:8->4
        heap: b:4 b:8 e:9 e:10
step 8  pop b:4  add ab(4)   total 28  updates -
        heap: b:8 e:9 e:10
      pop b:8  stale, discarded
step 9  pop e:9  add de(9)   total 37  updates -
        heap: e:10
      pop e:10  stale, discarded
tree: ha(8) ab(4) ic(2) cd(7) de(9) cf(4) fg(2) gh(1)
total weight 37; heap pushes 13, pops 13, stale pops 4
```

Answer. Thirteen pushes, thirteen pops, four stale; tree $\{ab, bc, ci, cf, fg, gh, cd, de\}$ of weight 37; running time $O(E \lg V)$.

> **CHECK YOURSELF.** Run the lazy version from i instead of a (the second half of the output above). Is the tree the same?
>
> *Answer.* The weight is again **37**, with four stale pops again, but the tree is **different**: it contains ha 8 instead of bc 8. From i the growing tree reaches h (by gh 1) before it reaches b, so a is first attached through h, at key 8, and b then comes in cheaply through ab 4, leaving bc unused. These are the two minimum spanning trees of §11.12: the tie between ah and bc, both of weight 8, is broken differently depending on where the tree starts.

12.6.5 Worked Problem U — the cut property and the cycle property

Problem. (a) State the cut property precisely, defining *cut*, *crossing edge*, *respects* and *light edge*, and prove it. (b) For the cut $S = \{A, B, C\}$ of G, list every crossing edge, name the light edge, and confirm that it is in the tree of Problem R. (c) State and justify the **cycle property**, and for each of the six non-tree edges of G name the cycle it closes and the heaviest tree edge on that cycle.

What it tests. The theorem that makes every minimum-spanning-tree algorithm correct (§11.7), and its mirror image.

Solution.

(a) The statement.

> **DEFINITIONS.** A **cut** $(S, V - S)$ is any partition of the vertices into two non-empty parts. An edge $\{u, v\}$ **crosses** the cut if exactly one of u, v lies in S. A set of edges A **respects** the cut if no edge of A crosses it. A crossing edge of minimum weight is a **light edge** for the cut (there may be several).
>
> **THE CUT PROPERTY (Theorem 11.5, in the form of §11.7.5).** Let A be a subset of the edges of some minimum spanning tree of G, let $(S, V - S)$ be any cut that A respects, and let (u, v) be a light edge for that cut. Then (u, v) is **safe** for A: that is, $A \cup \{(u, v)\}$ is also a subset of some minimum spanning tree.

The proof is an exchange argument, drawn in **Figure 12.20**. Take a minimum spanning tree T containing A and suppose the light edge (u, v) is not in it. Adding (u, v) to T creates exactly one cycle; that cycle must cross the cut a second time, at some edge (x, y). Since A respects the cut, $(x, y) \notin A$. Since (u, v) is light, $w(u, v) \leq w(x, y)$. So $T' = T - \{(x, y)\} + \{(u, v)\}$ is a spanning tree with $w(T') \leq w(T)$; as T is minimum, $w(T') = w(T)$ and T' is also minimum. T' contains A and (u, v). ■

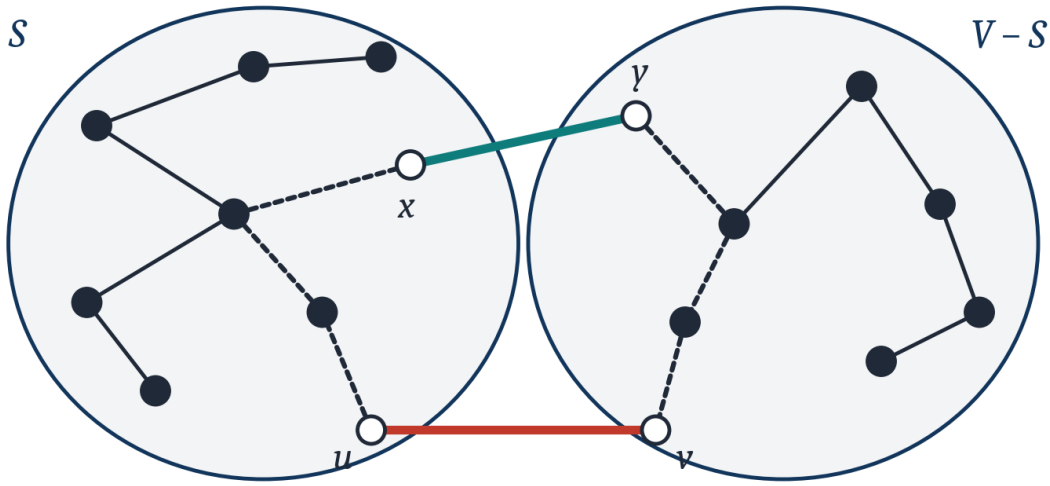


Figure 12.20 The exchange that proves the cut property. S and $V - S$ are the two blobs; the red edge (u, v) at the bottom is the light edge, assumed not to be in the tree; the dotted line is the unique tree path from u to v; and the teal edge (x, y) is the first edge of that path to cross the cut. Swapping (x, y) out for (u, v) is the whole proof.

(b) The cut $S = \{A, B, C\}$. Go through all thirteen edges and keep those with exactly one endpoint in S:

- AB — both in S. No. AC — both in. No. BC — both in. No.
- BD — B in, D out. **Crosses**, weight 6.
- CD — **crosses**, weight 4.

- CE — **crosses**, weight 9.
- $DE, DF, EF, EG, FG, FH, GH$ — both endpoints outside S. No.

Crossing edges: $\{BD\ 6,\ CD\ 4,\ CE\ 9\}$. The light edge is **CD** of weight 4, and **Table 12.31** shows it accepted at row 5. The cut property promised exactly that.

(c) The cycle property.

> **THE CYCLE PROPERTY.** If an edge e is the **unique** heaviest edge on some cycle of G, then e belongs to **no** minimum spanning tree. If it merely ties for heaviest, then some minimum spanning tree may contain it.

Why: let T be a minimum spanning tree containing such an e. Removing e splits T into two components; the cycle through e must contain another edge e' joining those two components, and $w(e') < w(e)$ by hypothesis. Then $T - e + e'$ is a spanning tree of strictly smaller weight — a contradiction.

Each non-tree edge closes exactly one cycle with the tree, namely the tree path between its endpoints plus itself; **Table 12.34** lists all six.

Table 12.34 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

Answer. The light edge of the cut $\{A, B, C\}$ is CD (4), and it is in the tree; every non-tree edge is strictly the heaviest on its cycle, so no minimum spanning tree contains any of them.

> **CHECK YOURSELF.** Take the cut $S = \{A, B, C, D\}$. Which edges cross it, and which is light?
>
> *Answer.* CE 9, DE 2 and DF 8 cross it. The light edge is DE, of weight 2, and it is in the tree — accepted at row 2 of **Table 12.31**.

12.6.6 Worked Problem V — union by rank and path compression

Problem. Starting from eight singletons $A, \ldots, H$, perform `Union(A,B)`, `Union(C,D)`, `Union(E,F)`, `Union(G,H)`, `Union(A,C)`, `Union(E,G)`, `Union(A,E)`, using **union by rank with path compression**. (a) Give the `parent` and `rank` arrays after each operation. (b) Then perform `Find` on every element and give the arrays again. (c) What is the greatest possible height of a tree on 8 elements under union by rank, and what would it be without it?

What it tests. The two tricks of §11.10.5 and §11.10.6, and the bound that makes them work.

Solution.

(a) The seven unions. `MakeSet` leaves every element its own parent with rank 0: `parent = ABCDEFGH`, `rank = 00000000`. Under union by rank, the root of *lower* rank is attached beneath the root of higher rank; on a tie, either may be chosen and the survivor's rank increases by one. The convention here attaches the second argument's root beneath the first's when the ranks tie.

Table 12.35 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

After the seventh union the forest is a single balanced tree of height 3: A at the root, with B, C and E beneath it; D beneath C; F and G beneath E; H beneath G:

```
      A
    / | \
  B   C   E
      |   | \
      D   F   G
              |
              H
```

(b) After a `Find` on every element. Each `Find` re-points every node on its path straight at the root. Performed in the order A to H: `Find(D)` walks $D \to C \to A$ and re-points D; `Find(F)` walks $F \to E \to A$ and re-points F; `Find(G)` walks $G \to E \to A$ and re-points G; and by the time `Find(H)` runs, G already points at A, so it walks $H \to G \to A$ and re-points H. The result is

`parent = AAAAAAAA`, `rank = 30102010`.

Every element now points directly at A, and every subsequent `Find` costs one step. The ranks are left **stale** — `rank[C]` still reads 1 although C is now a leaf — and that is correct behavior, not a bug: the rank is an upper bound on the height, and compression can only make the real height smaller.

Figure 12.21 shows the earlier, *list*-based representation of the same abstract structure (§11.10.3), and it is worth comparing: uniting two sets there costs time proportional to the size of the smaller list, because every element must be re-pointed. The forest of this problem does the same job in almost constant time by re-pointing only one root.

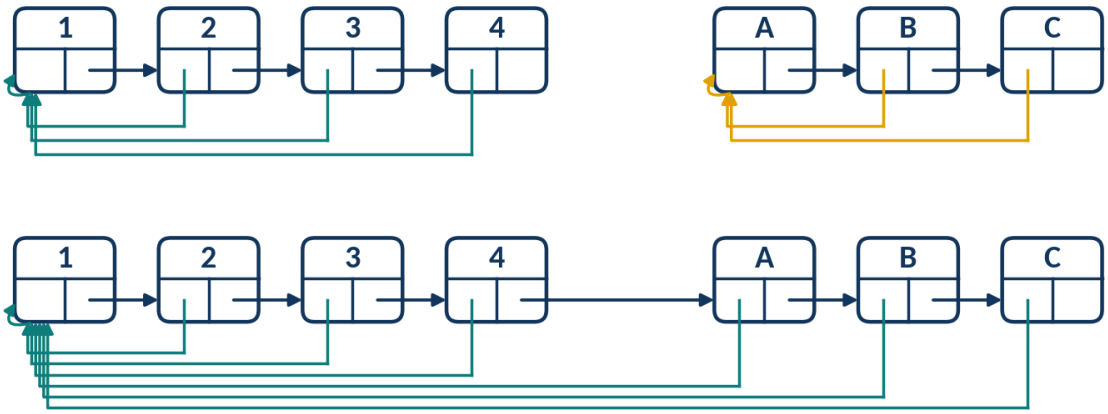


Figure 12.21 Disjoint sets as lists. **Top:** two sets, each element carrying a back-pointer to its representative. **Bottom:** the result of uniting them — the shorter list appended and every one of its elements re-pointed. Compare the cost with **Table 12.35**, where a union changes exactly one pointer.

(c) The heights. Under union by rank, a tree whose root has rank r contains at least 2^r elements:

$$size(x) \geq 2^{rank(x)} \qquad \text{for every root } x. \tag{12.19}$$

The proof of Equation (12.19) is by induction on the number of unions. It holds for singletons ($1 \geq 2^0$). A union of roots with different ranks leaves the larger rank unchanged and only adds elements, so the inequality survives. A union of two roots of equal rank r produces a root of rank $r+1$ whose tree contains at least $2^r + 2^r = 2^{r+1}$ elements. Since height never exceeds rank, a tree on n elements has height at most $\lfloor \lg n \rfloor$, and for $n = 8$ that is $\lg 8 = \mathbf{3}$ — which the run above attains exactly.

Without union by rank, attaching blindly can build a chain: `Union(A,B)`, `Union(B,C)`, …, `Union(G,H)` done naively gives a path of eight nodes, height **7**, and each `Find` then costs $\Theta(n)$. The measured difference is stark. On 100 000 elements chained the same way, a run with neither trick follows 4 999 950 000 parent pointers to perform one `Find` per element; with union by rank and path compression the same work costs 99 999 — a factor of fifty thousand (Exercise 12.11). The amortized bound behind this, $O(m\,\alpha(n))$ for m operations, is due to Tarjan [11].

Answer. The arrays of **Table 12.35**; after the finds, every parent is A and the ranks are unchanged; the height is at most 3 with union by rank and can be 7 without it.

> **CHECK YOURSELF.** Starting again from eight singletons, perform `Union(A,B)`, `Union(B,C)`, `Union(C,D)`, …, `Union(G,H)` — the chaining sequence — but *with* union by rank. What are the arrays at the end, and the height?
>
> *Answer.* The first union ties at rank 0, so B goes under A and `rank[A]` becomes 1. Every later union `Union(X,Y)` finds the root A (rank 1) for X and a singleton (rank 0) for Y, so Y goes directly under A. Final `parent = AAAAAAAA`, `rank = 10000000`, height **1**. Union by rank alone has defeated the chain, before path compression has done anything.

12.6.7 Worked Problem W — is the minimum spanning tree unique?

Problem. (a) Is the minimum spanning tree of G unique? Justify by deleting each tree edge in turn and recomputing. (b) Two edges of G have the same weight. Does that make the tree non-unique? Explain. (c) Lower $w(AC)$ from 5 to 3 and answer (a) again.

What it tests. §11.12: the difference between "ties exist" and "the tree is not unique".

Solution.

(a) The test. An edge-deletion test is the direct one: for each tree edge e, delete it from G and recompute the minimum spanning tree. If the new tree weighs the same as the old, an alternative minimum spanning tree exists that avoids e; if every deletion makes things strictly worse, the tree is unique.

Table 12.36 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

Seven rows of **Table 12.36**, seven strict increases, so the tree is unique. Note the last row: deleting GH forces the only other route to H, namely FH at 12, raising the total by exactly $12 - 11 = 1$. The cycle-property table **Table 12.34** proved the same thing more cheaply, in one pass.

(b) Equal weights. AB and FG both weigh 3, and the tree is nevertheless unique. So **repeated weights do not imply a non-unique tree.** The correct statements are:

- If all edge weights are **distinct**, the minimum spanning tree is unique (§11.12.2). Distinctness is *sufficient*.
- It is not *necessary*. What is both necessary and sufficient is the cycle-property condition of **Table 12.34**: the tree is unique exactly when every non-tree edge is the **strictly** heaviest edge on the cycle it closes.

Here AB and FG tie, but they lie in different parts of the graph and no cycle contains both, so the tie never forces a choice.

(c) Lowering $w(AC)$ to 3. Now the triangle A–B–C carries weights AB 3, BC 1, AC 3. The minimum spanning tree still weighs **31** — Kruskal now meets AB and AC together at weight 3, takes one of them, and rejects the other — but the choice is genuinely free. Deleting AB and recomputing still gives 31, with AC taking its place. So there are now **two** minimum spanning trees:

$$\{BC, DE, AB, FG, CD, EF, GH\} \quad \text{and} \quad \{BC, DE, AC, FG, CD, EF, GH\},$$

both of weight 31. The cycle-property test agrees: AC is no longer *strictly* heaviest on A–B–C–A; it ties with AB.

Answer. Unique, because every deletion makes the tree heavier; repeated weights do not by themselves break uniqueness; with $w(AC) = 3$ there are exactly two minimum spanning trees.

> **CHECK YOURSELF.** Restore $w(AC) = 5$, and instead lower $w(FH)$ from 12 to 11. Is the tree still unique?
>
> *Answer.* **No.** FH closes the cycle F–G–H–F, whose heaviest tree edge is GH at 11. Now FH also weighs 11: it ties rather than being strictly heaviest, so $\{BC, DE, AB, FG, CD, EF, FH\}$ is a second minimum spanning tree of weight 31. Counting every spanning tree by brute force finds exactly two of weight 31.

12.6.8 Problems R, S, U, V and W as one program

Listing 12.10 runs Problems R, S, U, V and W, printing the disjoint-set forest after every Kruskal decision and the waiting vertices after every Prim extraction.

Listing 12.10 Worked Problems R, S, U, V and W: the Kruskal trace with the disjoint-set forest, the Prim trace with the key array, the cut and cycle properties, the union-by-rank sequence, and the uniqueness test run twice. Compile with `g++ -std=c++17 -O2 -Wall -o mst mst.cpp`.

```
// Worked Problems R, S, U, V and W of Chapter 12 -- minimum spanning trees.
//   R  Kruskal, with the disjoint-set forest printed after every decision
//   S  Prim from A, with the key array printed after every extraction
//   U  the cut property and the cycle property, checked edge by edge
//   V  union by rank and path compression, by hand and by machine
//   W  is the minimum spanning tree unique?
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <numeric>
#include <climits>
using namespace std;

static const string NM = "ABCDEFGH";
struct E { int a, b, w; };
static const vector<E> G = {
    {0,1,3},{0,2,5},{1,2,1},{1,3,6},{2,3,4},{2,4,9},{3,4,2},
    {3,5,8},{4,5,7},{4,6,10},{5,6,3},{5,7,12},{6,7,11}
};
static const int N = 8;
static string nm(const E& e) { return string(1, NM[e.a]) + "-" + string(1, NM[e.b]); }

struct DSU {
    vector<int> p, r;
    explicit DSU(int n) : p(n), r(n, 0) { iota(p.begin(), p.end(), 0); }
    int find(int x) { return p[x] == x ? x : p[x] = find(p[x]); }
    bool unite(int x, int y) {
        x = find(x); y = find(y);
        if (x == y) return false;
        if (r[x] < r[y]) swap(x, y);
        p[y] = x; if (r[x] == r[y]) ++r[x];
        return true;
    }
    string show() {
        string s = "p:"; for (int i = 0; i < (int)p.size(); ++i) s += string(1, NM[p[i]]);
        s += "  r:"; for (int x : r) s += to_string(x);
        return s;
    }
};
```

Listing 12.10 (Continued) *Worked Problems R, S, U, V and W: the Kruskal trace with the disjoint-set forest, the Prim trace with the key array, the cut and cycle properties, the union-by-rank sequence, and the uniqueness test run twice. Compile with g++ -std=c++17 -O2 -Wall -o mst mst.cpp.*

```
vector<E> sorted_edges(vector<E> e) {
    sort(e.begin(), e.end(), [](const E& x, const E& y) {
        if (x.w != y.w) return x.w < y.w;
        if (x.a != y.a) return x.a < y.a;
        return x.b < y.b; });
    return e;
}

int kruskalWeight(const vector<E>& edges, int& taken) {
    DSU d(N); int t = 0; taken = 0;
    for (const E& e : sorted_edges(edges)) if (d.unite(e.a, e.b)) { t += e.w; ++taken; }
    return t;
}

int main() {
    cout << "=== Problem R  Kruskal ===\n";
    vector<E> S = sorted_edges(G);
    cout << "  sorted edge list:";
    for (const E& e : S) cout << " " << nm(e) << "(" << e.w << ")";
    cout << "\n";
    DSU d(N);
    int total = 0, taken = 0; vector<E> mst;
    cout << "  step  edge   w  find(a) find(b)  verdict   disjoint-set forest\n";
    int step = 0;
    for (const E& e : S) {
        int fa = d.find(e.a), fb = d.find(e.b);
        bool ok = d.unite(e.a, e.b);
        if (ok) { total += e.w; ++taken; mst.push_back(e); }
        cout << setw(6) << ++step << "  " << setw(4) << nm(e) << setw(4) << e.w
             << setw(8) << NM[fa] << setw(8) << NM[fb] << "  "
             << left << setw(10) << (ok ? "ACCEPT" : "reject") << right
             << d.show() << "\n";
        if (taken == N - 1) { cout << "  |V|-1 = " << N-1 << " edges taken; the remaining "
                                   << S.size() - step << " edges cannot be needed\n";
break; }
    }
    cout << "  MST:"; for (const E& e : mst) cout << " " << nm(e) << "(" << e.w << ")";
    cout << "\n  total weight " << total << "\n";
```

Listing 12.10 (Continued) *Worked Problems R, S, U, V and W: the Kruskal trace with the disjoint-set forest, the Prim trace with the key array, the cut and cycle properties, the union-by-rank sequence, and the uniqueness test run twice. Compile with g++ -std=c++17 -O2 -Wall -o mst mst.cpp.*

```
    cout << "\n=== Problem S  Prim from A ===\n";
    vector<vector<pair<int,int>>> adj(N);
    for (const E& e : G) { adj[e.a].push_back({e.b, e.w}); adj[e.b].push_back({e.a, e.w});
}
    for (auto& l : adj) sort(l.begin(), l.end());
    vector<int> key(N, INT_MAX), pi(N, -1); vector<char> inQ(N, 1);
    key[0] = 0;
    int ptotal = 0; vector<string> ptree;
    cout << "  step  extract  edge committed  running total  queue (v:key/pi)\n";
    for (int s = 1; s <= N; ++s) {
        int u = -1;
        for (int v = 0; v < N; ++v) if (inQ[v] && (u == -1 || key[v] < key[u])) u = v;
        inQ[u] = 0; ptotal += key[u];
        string ed = (pi[u] == -1) ? "(root)"
                  : string(1, NM[pi[u]]) + "-" + string(1, NM[u]) + "(" + to_string(key[u])
+ ")";
        if (pi[u] != -1) ptree.push_back(string(1, NM[pi[u]]) + "-" + string(1, NM[u]));
        for (auto [v, w] : adj[u]) if (inQ[v] && w < key[v]) { key[v] = w; pi[v] = u; }
        string q;
        vector<int> rest;
        for (int v = 0; v < N; ++v) if (inQ[v]) rest.push_back(v);
        sort(rest.begin(), rest.end(), [&](int x, int y){ return key[x] < key[y]; });
        for (int v : rest) q += string(1, NM[v]) + ":" +
            (key[v] == INT_MAX ? "inf" : to_string(key[v])) +
            (pi[v] == -1 ? "" : "/" + string(1, NM[pi[v]])) + " ";
        cout << setw(6) << s << setw(9) << NM[u] << setw(16) << ed
             << setw(15) << ptotal << "  " << (q.empty() ? "(empty)" : q) << "\n";
    }
    cout << "  Prim's tree:"; for (const string& e : ptree) cout << " " << e;
    cout << "\n  total weight " << ptotal
         << (ptotal == total ? "  -- the same as Kruskal's\n" : "  -- DISAGREES\n");

    cout << "\n=== Problem U  the cut property and the cycle property ===\n";
    vector<int> Sset = {0, 1, 2};                           // the cut S = {A,B,C}
    cout << "  cut S = {A,B,C}, V-S = {D,E,F,G,H}\n  crossing edges:";
    int light = INT_MAX; string le;
    for (const E& e : G) {
        bool ina = find(Sset.begin(), Sset.end(), e.a) != Sset.end();
        bool inb = find(Sset.begin(), Sset.end(), e.b) != Sset.end();
        if (ina != inb) { cout << " " << nm(e) << "(" << e.w << ")";
                          if (e.w < light) { light = e.w; le = nm(e); } }
    }
    cout << "\n  the light edge is " << le << " of weight " << light << "\n";
    bool inMst = false; for (const E& e : mst) if (nm(e) == le) inMst = true;
    cout << "  is it in the MST found in Problem R? " << (inMst ? "yes" : "no")
         << "  -- exactly as Theorem 11.5 promises\n";
    cout << "  the cycle property, checked on every non-tree edge:\n";
    for (const E& e : G) {
```

Listing 12.10 (Continued) *Worked Problems R, S, U, V and W: the Kruskal trace with the disjoint-set forest, the Prim trace with the key array, the cut and cycle properties, the union-by-rank sequence, and the uniqueness test run twice. Compile with g++ -std=c++17 -O2 -Wall -o mst mst.cpp.*

```
        bool tree = false; for (const E& t : mst) if (t.a == e.a && t.b == e.b) tree =
true;
        if (tree) continue;
        vector<vector<int>> t(N);
        for (const E& x : mst) { t[x.a].push_back(x.b); t[x.b].push_back(x.a); }
        vector<int> par(N, -2); queue<int> Q; Q.push(e.a); par[e.a] = -1;
        while (!Q.empty()) { int u = Q.front(); Q.pop();
            for (int v : t[u]) if (par[v] == -2) { par[v] = u; Q.push(v); } }
        vector<int> path; for (int x = e.b; x != -1; x = par[x]) path.push_back(x);
        int heaviest = 0; string ps;
        for (size_t i = 0; i + 1 < path.size(); ++i) {
            for (const E& x : mst)
                if ((x.a == path[i] && x.b == path[i+1]) || (x.b == path[i] && x.a ==
path[i+1]))
                    heaviest = max(heaviest, x.w);
            ps += string(1, NM[path[i]]) + "-";
        }
        ps += string(1, NM[path.back()]);
        cout << "    " << nm(e) << "(" << e.w << "): closes the cycle " << ps
             << ", whose heaviest tree edge weighs " << heaviest
             << "  -> " << (e.w > heaviest ? "strictly heaviest, so in NO MST"
                                          : "ties, so a second MST exists") << "\n";
    }

    cout << "\n=== Problem V  union by rank and path compression ===\n";
    DSU u1(8);
    vector<pair<int,int>> ops = {{0,1},{2,3},{4,5},{6,7},{0,2},{4,6},{0,4}};
    cout << "  operation   result   forest\n";
    for (auto [x, y] : ops) {
        bool ch = u1.unite(x, y);
        cout << "  Union(" << NM[x] << "," << NM[y] << ")   "
             << left << setw(9) << (ch ? "merged" : "no-op") << right << u1.show() << "\n";
    }
    cout << "  Find on every element, to show path compression:\n    ";
    for (int i = 0; i < 8; ++i) cout << NM[i] << "->" << NM[u1.find(i)] << " ";
    cout << "\n    " << u1.show() << "\n";
    cout << "  the tallest possible tree on 8 elements with union by rank has height "
         << 3 << " = lg 8; without it, a chain of 8 would have height 7\n";
```

Listing 12.10 (Continued) *Worked Problems R, S, U, V and W: the Kruskal trace with the disjoint-set forest, the Prim trace with the key array, the cut and cycle properties, the union-by-rank sequence, and the uniqueness test run twice. Compile with g++ -std=c++17 -O2 -Wall -o mst mst.cpp.*

```
    cout << "\n=== Problem W  is the minimum spanning tree unique? ===\n";
    int t0; int w0 = kruskalWeight(G, t0);
    cout << "  the MST weighs " << w0 << "\n";
    int alt = 0;
    for (const E& e : mst) {
        vector<E> without;
        for (const E& x : G) if (!(x.a == e.a && x.b == e.b)) without.push_back(x);
        int tk; int w = kruskalWeight(without, tk);
        cout << "    delete " << nm(e) << "(" << e.w << "): best spanning tree now weighs
";
        if (tk != N - 1) cout << "-- disconnected --\n";
        else { cout << w << (w == w0 ? "   <= SAME, so a second MST exists\n" : "
heavier\n");
               if (w == w0) ++alt; }
    }
    cout << "  rows with the same weight: " << alt
         << (alt == 0 ? "  -> the MST is UNIQUE\n" : "  -> the MST is NOT unique\n");
    cout << "  note: the weights are NOT all distinct (A-B and F-G both weigh 3),\n"
            "  and yet the tree is unique. Distinct weights are sufficient, not
necessary.\n";
    vector<E> G2 = G; for (E& e : G2) if (e.a == 0 && e.b == 2) e.w = 3;   // A-C: 5 -> 3
    int t2; int w2 = kruskalWeight(G2, t2);
    cout << "  now lower A-C from 5 to 3: the MST weighs " << w2 << "; deleting each tree
edge:\n";
    DSU dd(N); vector<E> mst2;
    for (const E& e : sorted_edges(G2)) if (dd.unite(e.a, e.b)) mst2.push_back(e);
    int alt2 = 0;
    for (const E& e : mst2) {
        vector<E> without;
        for (const E& x : G2) if (!(x.a == e.a && x.b == e.b)) without.push_back(x);
        int tk; int w = kruskalWeight(without, tk);
        if (tk == N - 1 && w == w2) { ++alt2;
            cout << "    dropping " << nm(e) << " still gives " << w << " -- a second
MST\n"; }
    }
    cout << "  alternatives found: " << alt2 << "\n";
    return 0;
}
```

The complete output reproduces every table above, including the subtlety of row 11 of **Table 12.31**:

```
=== Problem R  Kruskal ===
  sorted edge list: B-C(1) D-E(2) A-B(3) F-G(3) C-D(4) A-C(5) B-D(6) E-F(7) D-F(8) C-E(9)
E-G(10) G-H(11) F-H(12)
  step  edge   w  find(a) find(b)  verdict   disjoint-set forest
     1   B-C   1       B       C  ACCEPT    p:ABBDEFGH  r:01000000
     2   D-E   2       D       E  ACCEPT    p:ABBDDFGH  r:01010000
     3   A-B   3       A       B  ACCEPT    p:BBBDDFGH  r:01010000
     4   F-G   3       F       G  ACCEPT    p:BBBDDFFH  r:01010100
     5   C-D   4       B       D  ACCEPT    p:BBBBDFFH  r:02010100
     6   A-C   5       B       B  reject    p:BBBBDFFH  r:02010100
     7   B-D   6       B       B  reject    p:BBBBDFFH  r:02010100
     8   E-F   7       B       F  ACCEPT    p:BBBBBBFH  r:02010100
     9   D-F   8       B       B  reject    p:BBBBBBFH  r:02010100
    10   C-E   9       B       B  reject    p:BBBBBBFH  r:02010100
    11   E-G  10       B       B  reject    p:BBBBBBBH  r:02010100
    12   G-H  11       B       H  ACCEPT    p:BBBBBBBB  r:02010100
  |V|-1 = 7 edges taken; the remaining 1 edges cannot be needed
  MST: B-C(1) D-E(2) A-B(3) F-G(3) C-D(4) E-F(7) G-H(11)
  total weight 31

=== Problem S  Prim from A ===
  step  extract  edge committed  running total  queue (v:key/pi)
     1        A          (root)              0  B:3/A C:5/A D:inf E:inf F:inf G:inf H:inf
     2        B          A-B(3)              3  C:1/B D:6/B E:inf F:inf G:inf H:inf
     3        C          B-C(1)              4  D:4/C E:9/C F:inf G:inf H:inf
     4        D          C-D(4)              8  E:2/D F:8/D G:inf H:inf
     5        E          D-E(2)             10  F:7/E G:10/E H:inf
     6        F          E-F(7)             17  G:3/F H:12/F
     7        G          F-G(3)             20  H:11/G
     8        H         G-H(11)             31  (empty)
  Prim's tree: A-B B-C C-D D-E E-F F-G G-H
  total weight 31  -- the same as Kruskal's

=== Problem U  the cut property and the cycle property ===
  cut S = {A,B,C}, V-S = {D,E,F,G,H}
  crossing edges: B-D(6) C-D(4) C-E(9)
  the light edge is C-D of weight 4
  is it in the MST found in Problem R? yes  -- exactly as Theorem 11.5 promises
  the cycle property, checked on every non-tree edge:
    A-C(5): closes the cycle C-B-A, whose heaviest tree edge weighs 3  -> strictly
heaviest, so in NO MST
    B-D(6): closes the cycle D-C-B, whose heaviest tree edge weighs 4  -> strictly
heaviest, so in NO MST
    C-E(9): closes the cycle E-D-C, whose heaviest tree edge weighs 4  -> strictly
heaviest, so in NO MST
    D-F(8): closes the cycle F-E-D, whose heaviest tree edge weighs 7  -> strictly
heaviest, so in NO MST
    E-G(10): closes the cycle G-F-E, whose heaviest tree edge weighs 7  -> strictly
heaviest, so in NO MST
    F-H(12): closes the cycle H-G-F, whose heaviest tree edge weighs 11  -> strictly
heaviest, so in NO MST
```

```
=== Problem V  union by rank and path compression ===
  operation   result   forest
  Union(A,B)   merged   p:AACDEFGH  r:10000000
  Union(C,D)   merged   p:AACCEFGH  r:10100000
  Union(E,F)   merged   p:AACCEEGH  r:10101000
  Union(G,H)   merged   p:AACCEEGG  r:10101010
  Union(A,C)   merged   p:AAACEEGG  r:20101010
  Union(E,G)   merged   p:AAACEEEG  r:20102010
  Union(A,E)   merged   p:AAACAEEG  r:30102010
  Find on every element, to show path compression:
    A->A B->A C->A D->A E->A F->A G->A H->A
    p:AAAAAAAA  r:30102010
  the tallest possible tree on 8 elements with union by rank has height 3 = lg 8; without
it, a chain of 8 would have height 7

=== Problem W  is the minimum spanning tree unique? ===
  the MST weighs 31
    delete B-C(1): best spanning tree now weighs 35   heavier
    delete D-E(2): best spanning tree now weighs 37   heavier
    delete A-B(3): best spanning tree now weighs 33   heavier
    delete F-G(3): best spanning tree now weighs 38   heavier
    delete C-D(4): best spanning tree now weighs 33   heavier
    delete E-F(7): best spanning tree now weighs 32   heavier
    delete G-H(11): best spanning tree now weighs 32   heavier
  rows with the same weight: 0  -> the MST is UNIQUE
  note: the weights are NOT all distinct (A-B and F-G both weigh 3),
  and yet the tree is unique. Distinct weights are sufficient, not necessary.
  now lower A-C from 5 to 3: the MST weighs 31; deleting each tree edge:
    dropping A-B still gives 31 -- a second MST
  alternatives found: 1
```

12.7 Quick Self-Check Drills

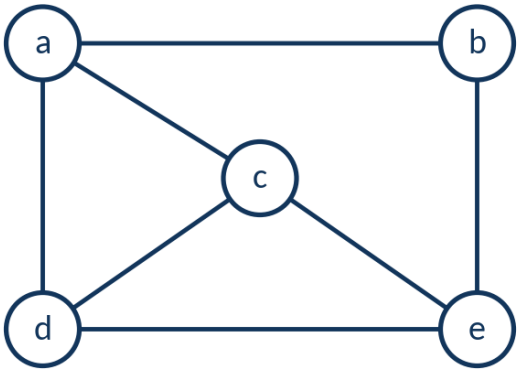
Thirty questions with one-line answers, covering every idea of Chapters 9, 10 and 11. Cover the answers, work down the list, and be honest with yourself. Anything you miss names the section to reread; the reference is given at the end of each answer, and **Table 12.1** points to the worked problem of the same kind.

1. **What two ingredients must a problem have before a greedy algorithm can be proved correct?** The greedy-choice property and optimal substructure. (§9.5)
2. **State the greedy-choice property in one sentence.** A globally optimal solution can be reached by making a locally optimal choice first — that is, *some* optimal solution contains the greedy choice. (§9.5.2)
3. **Maximal or maximum — what is the difference?** Maximal means nothing more can be added; maximum means nothing larger exists. Greed gives maximal for free; the whole proof burden is getting maximum. (§9.4.2)
4. **Which activity-selection rule is correct, and what does the algorithm cost?** Earliest finish time; $\Theta(n\lg n)$ to sort, then $\Theta(n)$ for one sweep — $\Theta(n)$ alone if the input is already sorted. (§9.4.12)
5. **Give a coin system and an amount on which the greedy change rule fails.** Coins $\{1,3,4\}$ and the amount 6: greed takes $4+1+1$ (three coins) where $3+3$ (two) is optimal. (§9.3.3)
6. **Which of the two ingredients does the 0-1 knapsack lack?** The greedy-choice property. It has optimal substructure, which is why dynamic programming works on it. (§9.6.4)
7. **What does Huffman's algorithm cost, and why?** $O(n\lg n)$: one heap build at $O(n)$, then $n-1$ iterations doing three heap operations each at $O(\lg n)$. (§9.7.11)

8. **Two independent ways to compute $B(T)$?** Sum $f(c) \times$ (depth of c) over the leaves; or sum the labels of all internal nodes. They must agree. (§9.7.5)
9. **The three conditions in the definition of a matroid?** S finite; $\mathcal{I}$ hereditary (subsets of independent sets are independent); the exchange property (if $|A| < |B|$ with both independent, some $x \in B \setminus A$ makes $A \cup \{x\}$ independent). (§9.8.2)
10. **In unit-time scheduling with deadlines and penalties, which sets of tasks are independent?** Those that can all be scheduled early — equivalently, those for which $N_t(A) \leq t$ for every t, where $N_t(A)$ counts tasks of A with deadline at most t. (§9.8.6)
11. **State the handshake lemma.** $\sum_v \deg(v) = 2|E|$, because every edge contributes one to the degree of each of its two endpoints. (§10.4.3)
12. **Space for an adjacency list against an adjacency matrix?** $\Theta(V + E)$ against $\Theta(V^2)$. On a million vertices the matrix is 10^{12} entries and the list may be 10^6. (§10.5.5)
13. **Running time of BFS and of DFS?** Both $\Theta(V + E)$ over an adjacency list, and $\Theta(V^2)$ over an adjacency matrix. (§10.7.7, §10.8.7)
14. **What three things does BFS compute?** The distance $d[v]$ from the source, the predecessor $\pi[v]$, and — by which vertices have finite d — the set reachable from the source. (§10.7.2)
15. **Why are BFS distances shortest-path distances?** Vertices leave the queue in non-decreasing order of d and every vertex is discovered from a neighbor one level closer, so $d[v] = \delta(s, v)$ by induction on δ. (§10.7.9)
16. **The three DFS colors, in one line.** White: undiscovered. Gray: discovered, still on the recursion stack. Black: finished, its whole subtree explored. (§10.8.2)
17. **State the parenthesis theorem.** For any two vertices, the intervals $[d, f]$ are either wholly nested or wholly disjoint; they never partially overlap. Nesting means ancestry. (§10.8.8)
18. **State the white-path theorem.** v becomes a descendant of u in the depth-first forest if and only if, at the moment u is discovered, v is reachable from u along a path of entirely white vertices. (§10.8.9)
19. **The four edge classes, and which can occur in an undirected graph?** Tree, back, forward, cross. In an undirected graph only tree and back edges occur. (§10.8.10)
20. **What does it cost to answer "are u and v connected?" for many pairs?** $\Theta(V + E)$ once to label every vertex with its component, then $O(1)$ per query. (§10.9.1)
21. **State the bipartiteness criterion.** A graph is bipartite if and only if it contains no odd cycle; equivalently, if and only if a two-coloring by BFS-distance parity succeeds. (§10.9.3)
22. **State the DAG theorem.** A directed graph is acyclic if and only if a depth-first search of it produces no back edge. (§11.2.3)
23. **The two topological-sort algorithms and their costs?** Depth-first search with vertices listed in decreasing finishing time, and Kahn's repeated removal of in-degree-zero vertices. Both $\Theta(V + E)$. (§11.3)
24. **Why decreasing f and not increasing d?** Because for every arc $u \to v$ in a DAG, $f[v] < f[u]$ always, whereas $d[v] < d[u]$ can fail. (§11.3.6)
25. **How many edges has a spanning tree, and how many spanning trees has K_n?** $|V| - 1$ edges; n^{n-2} spanning trees, by Cayley's formula. (§11.4)
26. **State the cut property.** If A lies in some minimum spanning tree and the cut $(S, V - S)$ respects A, then any light edge crossing that cut is safe for A. (§11.7.2)
27. **What is a safe edge?** One that can be added to the current set while keeping it a subset of some minimum spanning tree. (§11.8.1)

28. **Prim's cost against Kruskal's?** Prim: $O(E\lg V)$ with a binary heap, $O(E + V\lg V)$ with a Fibonacci heap. Kruskal: $O(E\lg E) = O(E\lg V)$, dominated by the sort. (§11.9.7, §11.10.12)
29. **What do union by rank and path compression buy?** Any sequence of m operations on n elements costs $O(m\,\alpha(n))$, and $\alpha(n) \leq 4$ for every n that will ever be run. (§11.10.7)
30. **When is the minimum spanning tree unique?** Exactly when every non-tree edge is the *strictly* heaviest edge on the cycle it closes. Distinct weights are sufficient for this but not necessary. (§11.12, §12.6.7)

One figure is worth having in front of you while you work the list: **Figure 12.22** carries every word of the graph vocabulary behind drills 11 to 21.



$$V = \{a, b, c, d, e\}$$

$$E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$$

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.

12.8 Profitina in Practice

Profitina is an AI-visibility platform — GEO, *Generative Engine Optimization* — live in production at `app.profitina.com`. It scans a brand's website, scores how ready that site is to be *cited by an AI answer engine* from real signals such as content extractability, structured data and the technical credibility of the site, and then helps fix what it finds. The design project of §12.2 asked you to find a real problem and recognize a taught algorithm inside it. That is the ordinary daily work of building software, and a scan of a website turns out to be built from exactly the algorithms this chapter has been working with. What follows describes the shape of each problem and the class of solution, not the product's proprietary scoring.

The crawl is a breadth-first search. A scan starts at the home page and follows links outwards. The number of clicks from the home page to a page — its **click depth** — is a real signal, because a page buried six clicks down is a page a crawler may never budget the time to reach. Click depth is precisely $d[v]$ from a breadth-first search over the link graph, and it must be computed with BFS rather than DFS: a depth-first crawl would wander to depth forty down a pagination chain and report absurd depths for pages that are one click from the front door. The same pass answers a second question for nothing: any page with $d[v] = \infty$ is an **orphan**, linked from nowhere, and orphan pages are invisible to crawler and answer engine alike.

Near-duplicate detection is a minimum spanning tree with the heavy edges cut. Sites accumulate pages that say almost the same thing — `/harga` and `/harga-2024`, three branch pages differing only in an address. An answer engine faced with several near-identical pages may cite none of them, or cite the wrong one. Measure a content distance between every pair of pages, build a minimum spanning tree over those distances, then **delete every tree edge heavier than**

a threshold: what remains is one component per cluster of near-duplicates. This is single-linkage clustering, and the reason it is a spanning-tree problem is the cut property of Problem U — the lightest edge leaving any group of pages is exactly the edge that decides which other group it should join first. **Figure 12.23** is the shape of the input: some edges are unavoidably expensive, some cheap-looking ones are redundant, and telling the two apart is the entire problem.

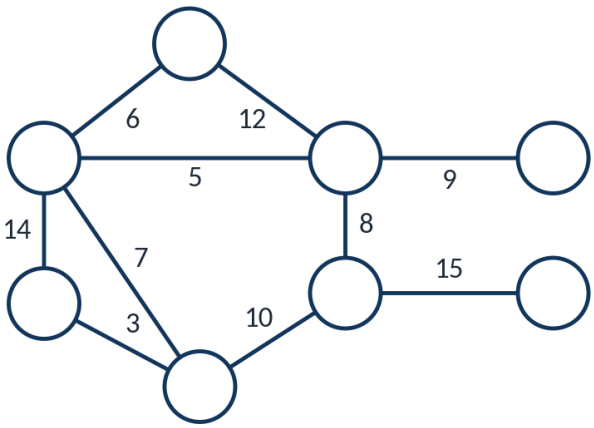


Figure 12.23 A weighted graph from Chapter 11. Some expensive edges are unavoidable — the 15 is the only way into its vertex — and some cheap-looking ones are redundant. A page-similarity graph has exactly this character, and the same algorithm separates the two kinds of edge.

The remediation plan is a DAG, and it is topologically sorted. Once a scan has produced a list of fixes, the fixes have precedences: canonical tags must be added before duplicate pages are merged, and the sitemap should not be resubmitted until `robots.txt` is correct. Run them in an order that violates those constraints and work is wasted. The order to run them in is a topological sort, the earliest possible completion is the critical path of Problem Q, and a circular dependency in the plan must be reported rather than silently half-executed — which is the argument of §12.2.5 for Kahn's algorithm, made in production.

Re-scans are a scheduling problem. The platform re-scans each brand automatically at a cadence set by its subscription tier, and alerts the owner when the score moves. At any moment many re-scans are due, each with a time by which it ought to have run; deciding which to run next is a priority-queue problem, and earliest-deadline-first — the rule Problem C proved optimal for the worst lateness — is the natural first rule to consider. Problem C also carries the warning: the right rule depends on the objective, and the rule that minimizes the *worst* delay is not the one that minimizes the *average*.

Two rules govern all of this, and both come from the product's own principles. First, it respects Indonesia's personal-data law, UU PDP 27/2022 [12]: a link graph and a page-similarity matrix are derived from public pages and touch no personal data. Second, **the app advises and the user decides** — the clustering proposes a canonical URL, it does not silently delete pages, and a score is an estimate from measured signals, never a guarantee of being cited.

12.8.1 The worked case study: Profitina Laundry

The runnable case study is always **Profitina Laundry**, the teaching business of this book, never the real product. The laundry now has a small website, and it wants to be found and cited when someone asks an answer engine *"where can I get laundry done near ITS?"*. **Listing 12.11** audits that website with three of the algorithms above.

Listing 12.11 The Profitina Laundry website audit: click depth and orphan pages by breadth-first search, near-duplicate clustering by a minimum spanning tree with the heavy edges cut, and the remediation plan topologically sorted with its critical path. Compile with `g++ -std=c++17 -O2 -Wall -o laundry laundry.cpp`.

```
// Profitina in Practice: auditing the Profitina Laundry website the way the
// AI-visibility engine audits a client's site. Three graph questions:
//   Part 1  click depth from the home page, by breadth-first search, and the
//           orphan pages nothing links to;
//   Part 2  near-duplicate page clusters, by building a minimum spanning tree
//           over content distances and cutting every edge above a threshold;
//   Part 3  the remediation plan, as a DAG, with a topological order.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <numeric>
#include <algorithm>
#include <climits>
using namespace std;

// ------------------------------------------------------ Part 1: the crawl
static const vector<string> URL = {
    "/", "/layanan", "/harga", "/cabang", "/cabang/keputih", "/cabang/gebang",
    "/tentang", "/kontak", "/blog", "/blog/merawat-kain", "/blog/tips-setrika",
    "/promo-lama"
};
static const vector<pair<int,int>> LINK = {
    {0,1},{0,2},{0,3},{0,6},{0,7},{0,8},{1,2},{3,4},{3,5},
    {8,9},{8,10},{9,10},{10,9},{4,7},{5,7},{11,0}
};

void part1() {
    int n = (int)URL.size();
    vector<vector<int>> out(n), in(n);
    for (auto [u,v] : LINK) { out[u].push_back(v); in[v].push_back(u); }
    vector<int> depth(n, INT_MAX);
    depth[0] = 0; queue<int> Q; Q.push(0);
    while (!Q.empty()) { int u = Q.front(); Q.pop();
        for (int v : out[u]) if (depth[v] == INT_MAX) { depth[v] = depth[u] + 1; Q.push(v);
} }
    cout << "=== Part 1: click depth from the home page ===\n";
    cout << left << setw(24) << "URL" << right << setw(8) << "depth"
         << setw(10) << "inlinks" << "  verdict\n";
    int deep = 0, orphan = 0;
    for (int v = 0; v < n; ++v) {
        string verdict = "ok";
        if (depth[v] == INT_MAX) { verdict = "ORPHAN - unreachable from /"; ++orphan; }
        else if (depth[v] > 2)   { verdict = "too deep for a crawler budget"; ++deep; }
        cout << left << setw(24) << URL[v] << right << setw(8);
        if (depth[v] == INT_MAX) cout << "inf"; else cout << depth[v];
```

Listing 12.11 (Continued) *The Profitina Laundry website audit: click depth and orphan pages by breadth-first search, near-duplicate clustering by a minimum spanning tree with the heavy edges cut, and the remediation plan topologically sorted with its critical path. Compile with g++ -std=c++17 -O2 -Wall -o laundry laundry.cpp.*

```
        cout << setw(10) << in[v].size() << "  " << verdict << "\n";
    }
    cout << "pages: " << n << ", orphans: " << orphan << ", deeper than 2 clicks: " << deep
<< "\n";
}

// ------------------------------------------ Part 2: near-duplicate clusters
struct D { int a, b, dist; };
static const vector<string> PAGE = {
    "/cabang/keputih", "/cabang/gebang", "/cabang/rungkut",
    "/harga", "/harga-2024", "/layanan"
};
static const vector<D> DIST = {                       // 0 = identical, 100 = unrelated
    {0,1,6},{0,2,9},{1,2,7},{0,3,64},{1,3,70},{2,3,66},
    {3,4,4},{0,4,61},{1,4,72},{2,4,68},{3,5,38},{4,5,41},
    {0,5,55},{1,5,58},{2,5,52}
};
struct DSU {
    vector<int> p, r;
    explicit DSU(int n) : p(n), r(n,0) { iota(p.begin(), p.end(), 0); }
    int find(int x) { return p[x]==x ? x : p[x]=find(p[x]); }
    bool unite(int x, int y) { x=find(x); y=find(y);
        if (x==y) return false;
        if (r[x]<r[y]) swap(x,y);
        p[y]=x; if (r[x]==r[y]) ++r[x];
        return true; }
};
void part2(int threshold) {
    cout << "\n=== Part 2: near-duplicate clusters (threshold " << threshold << ") ===\n";
    int n = (int)PAGE.size();
    vector<D> e = DIST;
    sort(e.begin(), e.end(), [](const D& x, const D& y){ return x.dist < y.dist; });
    DSU d(n); vector<D> mst; int total = 0;
    for (const D& x : e) if (d.unite(x.a, x.b)) { mst.push_back(x); total += x.dist; }
    cout << "  minimum spanning tree over " << n << " pages, " << DIST.size() << " measured
pairs:\n";
    for (const D& x : mst)
        cout << "    " << left << setw(18) << PAGE[x.a] << setw(18) << PAGE[x.b]
             << right << setw(5) << x.dist
             << (x.dist <= threshold ? "  keep (same cluster)" : "  CUT") << "\n";
    cout << "  tree weight " << total << "; edges " << mst.size() << " = " << n << " -
1\n";
    DSU c(n);
    for (const D& x : mst) if (x.dist <= threshold) c.unite(x.a, x.b);
    vector<vector<int>> cl(n);
    for (int v = 0; v < n; ++v) cl[c.find(v)].push_back(v);
    int k = 0;
    for (int v = 0; v < n; ++v) if (!cl[v].empty()) {
        cout << "  cluster " << ++k << ":";
```

Listing 12.11 (Continued) *The Profitina Laundry website audit: click depth and orphan pages by breadth-first search, near-duplicate clustering by a minimum spanning tree with the heavy edges cut, and the remediation plan topologically sorted with its critical path. Compile with g++ -std=c++17 -O2 -Wall -o laundry laundry.cpp.*

```
        for (int p : cl[v]) cout << " " << PAGE[p];
        cout << (cl[v].size() > 1 ? "   <- pick one canonical URL\n" : "\n");
    }
    cout << "  clusters: " << k << " (cutting the " << mst.size() - count_if(mst.begin(),
mst.end(),
            [&](const D& x){ return x.dist <= threshold; }) << " heaviest tree edges)\n";
}

// ------------------------------------------------ Part 3: the remediation DAG
static const vector<string> FIX = {
    "audit-inventory", "fix-robots.txt", "add-canonicals", "merge-duplicates",
    "add-schema.org", "rewrite-thin-pages", "submit-sitemap", "re-scan"
};
static const vector<int> HOURS = { 2, 1, 3, 4, 5, 8, 1, 1 };
static const vector<pair<int,int>> DEP = {
    {0,1},{0,2},{2,3},{0,5},{3,4},{5,4},{1,6},{3,6},{4,7},{6,7}
};
void part3() {
    cout << "\n=== Part 3: the remediation plan, topologically sorted ===\n";
    int n = (int)FIX.size();
    vector<vector<int>> adj(n); vector<int> indeg(n, 0);
    for (auto [u,v] : DEP) { adj[u].push_back(v); ++indeg[v]; }
    queue<int> Q; for (int v = 0; v < n; ++v) if (!indeg[v]) Q.push(v);
    vector<int> order, ef(n, 0), pred(n, -1);
    while (!Q.empty()) { int u = Q.front(); Q.pop(); order.push_back(u);
        if (!ef[u]) ef[u] = HOURS[u];
        for (int v : adj[u]) { if (ef[u] + HOURS[v] > ef[v]) { ef[v] = ef[u] + HOURS[v];
pred[v] = u; }
                              if (--indeg[v] == 0) Q.push(v); } }
    if ((int)order.size() != n) { cout << "  the plan contains a circular dependency\n";
return; }
    cout << "  step  task                 hours  earliest finish\n";
    for (size_t i = 0; i < order.size(); ++i)
        cout << setw(6) << i + 1 << "  " << left << setw(20) << FIX[order[i]]
             << right << setw(5) << HOURS[order[i]] << setw(16) << ef[order[i]] << "\n";
    int last = (int)(max_element(ef.begin(), ef.end()) - ef.begin());
    vector<int> path; for (int x = last; x != -1; x = pred[x]) path.push_back(x);
    reverse(path.begin(), path.end());
    cout << "  one worker, tasks end to end: " << accumulate(HOURS.begin(), HOURS.end(), 0)
         << " h\n  unlimited workers, critical path: " << ef[last] << " h  (";
    for (size_t i = 0; i < path.size(); ++i) cout << (i ? " -> " : "") << FIX[path[i]];
    cout << ")\n";
}

int main() { part1(); part2(30); part3(); return 0; }
```

The output:

```
=== Part 1: click depth from the home page ===
URL                          depth   inlinks  verdict
/                                0         1  ok
/layanan                         1         1  ok
/harga                           1         2  ok
/cabang                          1         1  ok
/cabang/keputih                  2         1  ok
/cabang/gebang                   2         1  ok
/tentang                         1         1  ok
/kontak                          1         3  ok
/blog                            1         1  ok
/blog/merawat-kain               2         2  ok
/blog/tips-setrika               2         2  ok
/promo-lama                    inf         0  ORPHAN - unreachable from /
pages: 12, orphans: 1, deeper than 2 clicks: 0

=== Part 2: near-duplicate clusters (threshold 30) ===
  minimum spanning tree over 6 pages, 15 measured pairs:
    /harga            /harga-2024           4  keep (same cluster)
    /cabang/keputih   /cabang/gebang        6  keep (same cluster)
    /cabang/gebang    /cabang/rungkut       7  keep (same cluster)
    /harga            /layanan             38  CUT
    /cabang/rungkut   /layanan             52  CUT
  tree weight 107; edges 5 = 6 - 1
  cluster 1: /cabang/keputih /cabang/gebang /cabang/rungkut   <- pick one canonical URL
  cluster 2: /harga /harga-2024   <- pick one canonical URL
  cluster 3: /layanan
  clusters: 3 (cutting the 2 heaviest tree edges)

=== Part 3: the remediation plan, topologically sorted ===
  step  task                 hours  earliest finish
     1  audit-inventory         2               2
     2  fix-robots.txt          1               3
     3  add-canonicals          3               5
     4  rewrite-thin-pages      8              10
     5  merge-duplicates        4               9
     6  add-schema.org          5              15
     7  submit-sitemap          1              10
     8  re-scan                 1              16
  one worker, tasks end to end: 25 h
  unlimited workers, critical path: 16 h  (audit-inventory -> rewrite-thin-pages -> add-
schema.org -> re-scan)
```

Three findings, in the language the laundry's owner speaks.

`/promo-lama` is invisible. It links *out* to the home page but nothing links *in*, so it has in-degree zero and infinite click depth. An answer engine will not find it, and neither will a customer. Either link to it from somewhere or retire it.

Five of the six audited pages are near-duplicates of something. Cutting the two tree edges above the threshold of 30 leaves three clusters: the three branch pages (distances 6 and 7 apart), the two pricing pages (distance 4), and `/layanan` alone. The recommendation is one canonical URL per cluster — not deletion, because the app advises and the owner decides.

The remediation takes 16 hours, not 25. Done strictly one task after another the plan is 25 hours; the critical path through the DAG is 16, because `fix-robots.txt`, `add-canonicals` and `rewrite-thin-pages` can all start as soon as the inventory is audited. All nine hours saved come from scheduling alone, obtained from a topological sort and one relaxation per arc.

12.9 Common Mistakes, and How to Avoid Them

Choosing a problem whose algorithm is decoration. A BFS on a board so small that any method would solve it. *Fix:* apply the load-bearing test of §12.2.2 before you commit to the idea — would the naive method survive a thousand-fold larger input?

Writing the design after the program. It then reads as description, not argument, and contains no rejected alternative. *Fix:* work stages 1 to 6 of **Table 12.2** first, and keep what you wrote.

Skipping the evaluation. A program run only on its example input has not been shown to work. *Fix:* a brute-force oracle or a postcondition, a few thousand random cross-checks, and timings at five sizes divided by the predicted cost, as **Listing 12.2** does.

Believing a greedy rule because it worked once. Three of the four activity rules give the optimum on the dryer jobs of Problem B, and two of them are wrong. *Fix:* either prove the rule by an exchange argument or break it with a counterexample, as Problem A does for coins; agreement on one instance proves nothing.

Writing "the optimal solution" in an exchange argument. The optimum is rarely unique, and the greedy-choice property says *some* optimal solution contains the greedy choice. *Fix:* write "let A be *a* maximum-size …".

Reading the matroid exchange property as "every element works". It promises that *some* $x \in B \setminus A$ can be added to A, not that all can; in Problem F, a_4 works and a_5 does not.

Writing down a Huffman code without checking it. A slip in one merge silently changes several codewords. *Fix:* compute $B(T)$ both ways, by Equation (12.9) and by Equation (12.10); if they disagree, there is an error.

Showing only the end result of a trace. The trace *is* the answer when the question asks for one; a final tree with no working shows nothing about whether `Find` and `Union` were understood. *Fix:* one row per step, every intermediate value written down, as in **Table 12.31**.

Applying the directed cycle test to an undirected graph. Every edge then looks like a cycle, because each edge sits in both adjacency lists. *Fix:* skip exactly one occurrence of the parent, and no more (Problem L).

Looking for forward or cross edges in an undirected depth-first search. There are none; a second sighting of an edge from its black end is the same back edge seen again (Problem I).

Calling a black-vertex arc "forward" because the target comes later. Forward means *descendant in the depth-first forest*; the timestamp test $d[u] < d[v]$ decides it, and a vertex in another tree always fails it (Problem M).

Sorting by discovery time to get a topological order. Only finishing time works, by Equation (12.15); Problem N has the two-vertex counterexample.

Forgetting the stale entries in a lazy heap. Without the test on line 5 of **Pseudocode 12.7**, Prim would process h three times in the run of Problem T and add its weight to the total three times. *Fix:* on every pop, discard the entry if its vertex is already in the tree or its key is out of date (Problem T).

Believing that equal weights make the minimum spanning tree non-unique. Distinctness is sufficient, not necessary; the true criterion is the cycle property (Problem W).

Believing that Prim and Kruskal must return the same edges. They must return the same *weight*. On a graph with tied weights they may well disagree about which edges (**Figure 12.18**, and the second run of Problem T).

Reporting the count of back edges as if it were order-independent. The *verdict* "not acyclic" is order-independent; the count and the classification are not (Exercise 12.5).

Treating `rank` as a height. It is an upper bound that path compression deliberately leaves stale (Problem R).

12.10 Chapter Summary

This chapter worked Chapters 9 to 11 as problems rather than as theory.

§12.2 took one real problem from a blank page to a measured program in nine stages. The Rute Laundry planner for Profitina Laundry modeled a road network and a processing pipeline as graphs, chose breadth-first search, Kruskal over a disjoint-set forest, and Kahn's algorithm against named alternatives, predicted a cost of $\Theta(m\lg m)$ in advance, and traced all three algorithms by hand before writing a line of code. The program then agreed with the hand traces line for line: six of fourteen zones within two hops and two unreachable, an eleven-road backbone of 6.9 km against 12.3 km of candidates (a 43.9% saving), and a 121-minute critical path with 106 minutes of slack on invoicing. The evaluation cross-checked Kruskal against Prim on 2000 random graphs with no mismatch, checked the BFS level postcondition on 2000 more with no violation, and measured both algorithms at five sizes against their predicted growth — and explained why the linear one stopped looking linear once the data left the cache.

§12.3 worked six greedy problems: a coin system on which greed fails, with a proof that it succeeds for {1,5,10}; activity selection run under two rules and proved by an exchange argument; minimizing maximum lateness, proved by exchanging adjacent inversions; a Huffman code built merge by merge, drawn as a tree and costed two ways; the two knapsacks, where one ingredient separates success from a 27.8% loss; and the matroid greedy algorithm scheduling unit-time orders, confirmed against all 128 subsets. §12.4 traced breadth-first search, depth-first search on a directed and on an undirected graph with full timestamp tables, components with the circuit rank, bipartiteness with its odd-cycle certificate, and both cycle tests. §12.5 worked the prerequisite DAG: the back-edge test, topological sort by finishing time and by Kahn, the 84 legal orders and the three extra prerequisites needed to force one, and the critical path. §12.6 worked minimum spanning trees: Kruskal with every `Find` and `Union`, Prim with a key array and with a lazy binary heap, the cut and cycle properties, union by rank with the 2^{rank} bound, and the test for uniqueness. Every problem ended with a variant to check yourself, and twelve complete programs confirmed every number.

12.10.1 Looking ahead

Every graph algorithm in this chapter treated its edges either as unweighted or as costs to be *summed over a tree*. Chapter 13 changes the question: given weights that are travel times, what is the cheapest **path** from one vertex to another? Breadth-first search answers it when every edge costs the same; Dijkstra's algorithm answers it when the costs are non-negative; Bellman–Ford answers it when some costs are negative; and on a DAG a single topological sweep — the same sweep that produced the critical paths of §12.2.8 and Problem Q — answers it in linear time. The conclusion of the design project already named that as its next step.

12.11 Exercises

12.1 For the coin system {1,4,6}, find the smallest amount on which the greedy change rule of **Pseudocode 12.2** fails, give both payments for it, and count the amounts from 1 to 30 on which greed loses. Is the smallest failure consistent with the bound of Kozen and Zaks quoted in §12.3.2?

12.2 Apply the load-bearing test of §12.2.2 to these three proposals, and for each say what the design stage would be able to argue. (a) A Sudoku solver that uses depth-first backtracking. (b) A campus map app that computes the fewest-turns route between two buildings. (c) A quiz app that shuffles ten questions.

12.3 Run breadth-first search on the graph G_1 of §12.4.1 from source C instead of A, giving d and π. Which vertices change their distance, and what are the greatest distances from A and from C?

12.4 In the Rute Laundry instance of §12.2.4, the operations team discovers a road from **Menur to Klampis**. Recompute the breadth-first distances from Sukolilo. How many zones now lie within the two-hop promise, and which zone joined it?

12.5 Run depth-first search on the digraph G_2 of §12.4.1 with the outer loop scanning G, F, E, D, C, B, A instead of A to G. Give the timestamps and classify every arc. Which of the following are unchanged: the number of tree edges, the number of back edges, the verdict "not acyclic"?

12.6 The depth-first search of the prerequisite DAG in Problem M produced five tree edges. Explain why that number had to be 8 minus the number of trees in the forest, and say what would change if the outer loop started at H.

12.7 Construct an instance of activity selection on which the *shortest duration first* rule returns a strictly smaller set than the earliest-finish rule. Use as few activities as you can, and state the answer each rule gives.

12.8 Suppose all n symbols in a Huffman instance have the **same** frequency f. (a) What shape is the tree, and what are the codeword lengths, when $n = 8$? When $n = 6$? (b) What is $B(T)$ compared with a fixed-length code in each case? (c) True or false: every Huffman tree on three or more symbols has a codeword of length 1. Prove it or give a counterexample, and state something that *is* true of every Huffman tree.

12.9 Take the graph G of §12.6.1. Compute the minimum spanning tree after each of these transformations and say whether the *tree* changes, not merely its weight: (a) add 10 to every weight; (b) square every weight; (c) replace every weight w by $20 - w$.

12.10 How many independent cycles does the graph G of §12.6.1 contain? Derive the number two ways: from Equation (12.14), and by counting the non-tree edges of **Table 12.34**.

12.11 Chain 100 000 elements together with `Union(0,1)`, `Union(1,2)`, …, and then perform one `Find` on each element. Count the total number of parent pointers followed, (a) with no union by rank and no path compression, and (b) with both. Explain the ratio.

12.12 In the prerequisite DAG of **Table 12.27**, Databases (F) is lengthened from 3 weeks to 7. Recompute every earliest finishing time and the critical path. By how much does the program lengthen, and which subject leaves the critical path?

12.13 Replace Kahn's **queue** by a **stack** (push the zero-in-degree vertices in alphabetical order, then repeatedly pop). Run it on the prerequisite DAG of **Table 12.27** and give the order. Compare it with the orders of Problems N and O.

12.14 For each of the three components of the graph G_3 of Problem J, say whether it is bipartite and give either the two sides or an odd cycle.

12.15 Take an idea of your own for a design project that passes the load-bearing test. In no more than 400 words, write stages 1 to 5 of **Table 12.2**: the problem, the model, the algorithm chosen with one rejected alternative and its reason, the pseudocode in outline, and the predicted complexity. Then check your draft against the third column of **Table 12.2**.

12.12 Solutions to the Exercises

12.1 From the verified run of **Listing 12.12**: the smallest failure is **8**, where greed pays $6 + 1 + 1$ (three coins) and the optimum is $4 + 4$ (two). Greed loses on **8** of the amounts from 1 to 30; the first four are 8, 9, 14 and 15 (at 9, greed pays $6 + 1 + 1 + 1$ against $4 + 4 + 1$). The Kozen–Zaks bound says that a first failure, if there is one, lies below the sum of the two largest coins, $6 + 4 = 10$; and indeed $8 < 10$.

12.2 (a) **Sudoku by backtracking.** A naive "try every filling of the empty cells" is up to 9^{81} and hopeless — but so is unpruned backtracking. This is load-bearing *only if* the design argues about the pruning (constraint propagation, most-constrained-cell ordering). Argue that, and it is a strong project. (b) **Fewest turns between two buildings.** Model the turns, not the roads: a vertex is a pair (junction, heading), and an edge that changes heading costs one. Then it is a breadth-first search on a graph with $2|E|$ vertices, and the naive method — enumerating routes — dies immediately. **Load-bearing**, and unusual enough that the modeling step itself is the interesting part of the design. (c) **Shuffling ten questions.** Fisher–Yates is $\Theta(n)$ and ten is ten. **Decoration**; there is nothing for the design to argue.

12.3 From the verified run of **Listing 12.12**, collected in **Table 12.37**:

Table 12.37 Breadth-first search of G_1 from C, beside the search from A. Every vertex changes its distance. The greatest distance to any vertex — the **eccentricity** of the source — is 4 from A and 3 from C, so C is the more central of the two and its tree is one level shallower.

v	A	B	C	D	E	F	G	H	I
d from A	0	1	1	2	2	2	3	3	4
d from C	1	2	0	3	1	1	2	2	3
π from C	C	A	–	B	C	C	E	E	G

12.4 Adding Menur–Klampis puts Klampis one hop from Menur, hence **two** hops from Sukolilo instead of three. The new distances are Sukolilo 0; Menur 1, Keputih 1; Gebang 2, Semolowaru 2, **Klampis 2**, Kejawan 2; Nginden 3, Manyar 3, Mulyosari 3; Kalijudan 4, DukuhSetro 4; Bulak and Kenjeran still unreachable. The two-hop count rises from 6 to **7**, and the zone that joined is **Klampis**. Note the knock-on effect: Manyar falls from 4 to 3 and DukuhSetro from 5 to 4, so one road bought improvements for three zones — but it bought nothing at all for Bulak and Kenjeran, because adding an edge inside one component can never create a path into another.

12.5 With the outer loop reversed the timestamps become G 1/14, C 2/13, D 3/8, A 4/7, B 5/6, E 9/12, F 10/11, and the classification becomes six tree edges ($G \to C$, $C \to D$, $D \to A$, $A \to B$, $C \to E$, $E \to F$), four back ($A \to C$, $A \to D$, $B \to D$, $F \to E$), one forward ($G \to F$) and **zero** cross edges.

So the number of tree edges **changed**, from 5 to 6, because the forest now has one tree instead of two ($7 - 1 = 6$). The number of back edges **changed**, from 2 to 4. The verdict "not acyclic" is **unchanged**, and that is the only one of the three that the DAG theorem guarantees: acyclicity is a property of the graph, while the classification is a property of the search.

12.6 Every vertex except a root acquires exactly one predecessor, and every tree edge is the pair $(\pi[v], v)$ for exactly one such vertex. So the count is $|V| - (\text{number of roots}) = 8 - 3 = 5$. The roots are the vertices the outer loop reaches while they are still white, which here are A, B and C. If the outer loop started at H, then H — which has no outgoing arcs — would finish immediately as a one-vertex tree, and the forest would gain a root: the number of roots would rise to four and the number of tree edges would fall to $8 - 4 = 4$, because the arc into H that used to be a tree edge $(E \to H)$ would now find H black and become a cross edge.

12.7 Three activities suffice: $a_1 = [0,5)$, $a_2 = [4,6)$, $a_3 = [5,10)$. Durations are 5, 2 and 5, so *shortest duration first* takes a_2, which overlaps both of the others, and stops with $\{a_2\}$, of size **1**. *Earliest finish time* takes a_1 (finishing at 5), then a_3 (starting at 5), giving $\{a_1, a_3\}$, of size **2**. Two is optimal. The instance is minimal: with two activities the two rules cannot differ in the size of the answer.

12.8 (a) With $n = 8$ equal frequencies every merge joins two trees of equal weight, so the tree is a **perfect binary tree** of depth 3 and every codeword has length $\lg 8 = 3$. With $n = 6$ the counts do not stay equal: merging two leaves of weight f makes a node of weight $2f$, so after three merges the queue holds three nodes of weight $2f$; two of them merge into $4f$, and the last merge joins $2f$ with $4f$. The tree therefore ends with **four** symbols at depth 3 and **two** at depth 2 — lengths $\lceil \lg 6 \rceil = 3$ and $\lfloor \lg 6 \rfloor = 2$. (b) For $n = 8$, $B(T) = 8f \times 3 = 24f$, exactly the fixed-length cost: **no saving at all**. For $n = 6$, $B(T) = 4f \times 3 + 2f \times 2 = 16f$ against a fixed-length $6f \times 3 = 18f$, a saving of $2f$, or 11.1%. (c) **False.** Part (a) is a counterexample: with eight equal frequencies every codeword has length 3, and none has length 1. A one-bit codeword appears only when one of the two trees joined at the last merge is a single leaf, which requires that symbol to be frequent relative to the rest, as R was in Problem D. What *is* true of every Huffman tree is that it is **full** — every internal node has exactly two children, so no bit is wasted — and that the two least frequent symbols are siblings at maximum depth (§9.7.8).

12.9 From the verified run: the original tree weighs 31 and is $\{BC, DE, AB, FG, CD, EF, GH\}$. (a) **Add 10:** weight $31 + 7 \times 10 = 101$, **same tree**. Adding a constant k adds exactly $k(|V| - 1)$ to *every* spanning tree, so the ranking is untouched. (b) **Square:** weight 209, **same tree**. Squaring is strictly increasing on non-negative weights, so it preserves the *order* of the weights, and both Kruskal and Prim depend only on that order. (It would not preserve the order if any weight were negative.) (c) $20 - w$**:** weight 79, and a **different tree** — $\{FH, GH, EG, CE, DF, BD, AC\}$, which is the *maximum* spanning tree of the original graph. Reversing the order of the weights turns the minimization into a maximization.

The lesson: a transformation that preserves the order of the weights leaves the tree alone; one that reverses the order turns it inside out.

12.10 By Equation (12.14), the circuit rank is $|E| - |V| + k = 13 - 8 + 1 = \mathbf{6}$. Counting the other way, a spanning tree uses $|V| - 1 = 7$ edges, so $13 - 7 = 6$ edges are left over, and each one closes exactly one new cycle when added back — those are precisely the six rows of **Table 12.34**. The two counts agree, as they must, because $|E| - (|V| - k)$ and $|E| - |V| + k$ are the same expression.

12.11 Measured by **Listing 12.12**: **(a)** with no union by rank and no path compression, the chain builds a path of 100 000 nodes and one `Find` per element follows $0 + 1 + 2 + \cdots + 99\,999 = \frac{99\,999 \times 100\,000}{2} = 4\,999\,950\,000$ parent pointers. **(b)** With union by rank and path compression, the total is 99 999 — one hop per element, because after the unions every element already points

directly at the root. The ratio is about **50 000 to 1**, and it grows linearly in n: the naive structure is $\Theta(n^2)$ over n finds, the good one $O(n\,\alpha(n))$.

12.12 With Databases at 7 weeks, $ef[F] = ef[D] + 7 = 8 + 7 = 15$, and then

$$ef[H] = \max\{ef[E], ef[F], ef[G]\} + 4 = \max\{14,15,6\} + 4 = \mathbf{19}.$$

The program lengthens from 18 weeks to 19, and the critical path moves from $A \to D \to E \to H$ to $A \to D \to F \to H$. **Algorithms (E) leaves the critical path**, acquiring one week of slack; Databases joins it. This is the characteristic behavior of critical paths: lengthening a task with slack does nothing until the slack is used up, and then the path jumps.

12.13 Pushing A, B, C and repeatedly popping gives: pop C (emit, push G); pop G (emit); pop B (emit); pop A (emit, push D); pop D (emit, push E then F); pop F (emit); pop E (emit, push H); pop H. Order:

$$C,\ G,\ B,\ A,\ D,\ F,\ E,\ H.$$

That is **identical to the depth-first order** of Problem N and completely different from the queue-based order of Problem O. The resemblance is not an accident of this instance alone: a stack makes Kahn's algorithm dive as deep as the in-degrees allow before backtracking, which is what depth-first search does. It is not a theorem, though — the two can differ when a vertex becomes ready long before it is reached — so on any given DAG the agreement must be checked, not assumed.

12.14 The **triangle** {1,2,3} is **not bipartite**: 1–2–3–1 is an odd cycle of length 3. The **star** {4,5,6,7} **is bipartite**, with sides {4} and {5,6,7}; every star is, since it is a tree, and every tree is bipartite (a tree has no cycle at all, so no odd one). The **five-cycle** {8,9,10,11,12} is **not bipartite**: 8–9–10–11–12–8 is an odd cycle of length 5. So G_3 as a whole is not bipartite, and a two-coloring fails in two of its three components independently.

12.15 There is no single answer; check your draft against the third column of **Table 12.2**. The draft is complete if a reader who has never seen your program can, from your 400 words alone: name the problem, name the input and its size, name the algorithm, say why the obvious alternative was rejected, and predict the running time. If any of those five is missing, add it — each takes one sentence. Then apply the load-bearing test once more to what you wrote: if the naive method you rejected would in fact survive a thousand-fold larger input, the argument for your algorithm needs a larger input, or a different problem.

12.12.1 The programs

Listing 12.12 computes every number quoted in the solutions to Exercises 12.1, 12.3, 12.5, 12.7, 12.9 and 12.11.

Listing 12.12 The programs for Exercises 12.1, 12.3, 12.5, 12.7, 12.9 and 12.11. Compile with `g++ -std=c++17 -O2 -Wall -o exercises exercises.cpp` and run `./exercises`.

```
// Programs for the exercises of Chapter 12.
//   Ex 12.1   the coin system {1, 4, 6}: where does greedy change-making fail?
//   Ex 12.3   BFS from a different source on the same graph
//   Ex 12.5   DFS with the outer loop reversed: a different forest, same graph
//   Ex 12.7   an activity instance on which "shortest duration first" fails
//   Ex 12.9   adding a constant to every weight leaves the MST alone;
//             squaring the weights may not
//   Ex 12.11  union-find with and without union by rank, measured
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <numeric>
#include <functional>
#include <climits>
using namespace std;

// ---------------------------------------------------------------- Ex 12.1
void ex1() {
    cout << "=== Ex 12.1  greedy change-making with the coins {1, 4, 6} ===\n";
    const vector<int> coins = {6, 4, 1};                    // largest first
    const int maxX = 30;
    vector<int> best(maxX + 1, INT_MAX); best[0] = 0;       // fewest coins, by DP
    for (int x = 1; x <= maxX; ++x)
        for (int c : coins) if (c <= x && best[x - c] + 1 < best[x]) best[x] = best[x - c]
+ 1;
    int bad = 0;
    for (int x = 1; x <= maxX; ++x) {
        int g = 0, r = x;
        for (int c : coins) { g += r / c; r %= c; }
        if (g > best[x]) {
            if (++bad <= 4) cout << "  amount " << setw(2) << x << ": greedy uses " << g
                                 << " coins, the optimum uses " << best[x] << "\n";
        }
    }
    cout << "  amounts 1.." << maxX << " on which greed loses: " << bad << "\n";
}
```

Listing 12.12 (Continued) *The programs for Exercises 12.1, 12.3, 12.5, 12.7, 12.9 and 12.11. Compile with g++ -std=c++17 -O2 -Wall -o exercises exercises.cpp and run ./exercises.*

```
// ---------------------------------------------------------------- Ex 12.3
static const string UN = "ABCDEFGHI";
static const vector<pair<int,int>> UE = {
    {0,1},{0,2},{1,3},{1,4},{2,4},{2,5},{3,6},{4,6},{4,7},{5,7},{6,8},{7,8}
};
void ex3() {
    cout << "\n=== Ex 12.3  BFS from C instead of A ===\n";
    int n = (int)UN.size();
    vector<vector<int>> adj(n);
    for (auto [a,b] : UE) { adj[a].push_back(b); adj[b].push_back(a); }
    for (auto& l : adj) sort(l.begin(), l.end());
    for (int s : {0, 2}) {
        vector<int> d(n, INT_MAX), pi(n, -1);
        d[s] = 0; queue<int> Q; Q.push(s);
        while (!Q.empty()) { int u = Q.front(); Q.pop();
            for (int v : adj[u]) if (d[v] == INT_MAX) { d[v] = d[u]+1; pi[v] = u;
Q.push(v); } }
        cout << "  source " << UN[s] << "   d: ";
        for (int v = 0; v < n; ++v) cout << UN[v] << "=" << d[v] << " ";
        cout << "\n            pi: ";
        for (int v = 0; v < n; ++v) cout << UN[v] << "<-" << (pi[v] < 0 ? '-' : UN[pi[v]])
<< " ";
        cout << "\n";
    }
    cout << "  the eccentricity of A is 4 and of C is 3, so the two trees differ in
shape\n";
}

// ---------------------------------------------------------------- Ex 12.5
static const string DN = "ABCDEFG";
static const vector<pair<int,int>> DE = {
    {0,1},{0,2},{0,3},{1,3},{2,3},{2,4},{3,0},{4,5},{5,4},{6,2},{6,5}
};
void dfsWithOrder(const vector<int>& order, const string& label) {
    int n = (int)DN.size();
    vector<vector<int>> adj(n);
    for (auto [u,v] : DE) adj[u].push_back(v);
    for (auto& l : adj) sort(l.begin(), l.end());
    vector<int> col(n,0), d(n,0), f(n,0), pi(n,-1); vector<string> cls(DE.size());
    int clk = 0;
    function<void(int)> visit = [&](int u) {
        col[u] = 1; d[u] = ++clk;
        for (int v : adj[u]) {
            int idx = -1;
            for (size_t i = 0; i < DE.size(); ++i)
                if (DE[i].first == u && DE[i].second == v) idx = (int)i;
            if (col[v] == 0) { cls[idx] = "T"; pi[v] = u; visit(v); }
            else if (col[v] == 1) cls[idx] = "B";
            else cls[idx] = (d[u] < d[v] ? "F" : "C");
```

Listing 12.12 (Continued) *The programs for Exercises 12.1, 12.3, 12.5, 12.7, 12.9 and 12.11. Compile with g++ -std=c++17 -O2 -Wall -o exercises exercises.cpp and run ./exercises.*

```
        }
        col[u] = 2; f[u] = ++clk;
    };
    for (int v : order) if (col[v] == 0) visit(v);
    cout << "  " << label << "\n    d/f: ";
    for (int v = 0; v < n; ++v) cout << DN[v] << "=" << d[v] << "/" << f[v] << " ";
    int t=0,b=0,fw=0,c=0;
    for (const string& s : cls) { if (s=="T") ++t; else if (s=="B") ++b; else if (s=="F")
++fw; else ++c; }
    cout << "\n    tree " << t << ", back " << b << ", forward " << fw << ", cross " << c
<< "\n    ";
    for (size_t i = 0; i < DE.size(); ++i)
        cout << DN[DE[i].first] << "->" << DN[DE[i].second] << ":" << cls[i] << " ";
    cout << "\n";
}
void ex5() {
    cout << "\n=== Ex 12.5  the same digraph, two outer-loop orders ===\n";
    vector<int> fwd(7); iota(fwd.begin(), fwd.end(), 0);
    vector<int> rev = fwd; reverse(rev.begin(), rev.end());
    dfsWithOrder(fwd, "outer loop A,B,C,D,E,F,G");
    dfsWithOrder(rev, "outer loop G,F,E,D,C,B,A");
    cout << "  the COUNT of back edges changed (2 then 4) but neither is zero:\n"
            "  the verdict 'not acyclic' is order-independent, the classification is
not\n";
}

// ---------------------------------------------------------------- Ex 12.7
struct Act { int i, s, f; };
bool compat(const Act& a, const Act& b) { return a.f <= b.s || b.f <= a.s; }
void ex7() {
    cout << "\n=== Ex 12.7  an instance where 'shortest duration first' fails ===\n";
    vector<Act> A = { {1,0,5}, {2,4,6}, {3,5,10} };
    cout << "  a1=[0,5), a2=[4,6), a3=[5,10)\n";
    vector<Act> left = A; vector<int> pick;
    while (!left.empty()) {
        const Act* best = &left[0];
        for (const Act& a : left) if (a.f - a.s < best->f - best->s) best = &a;
        Act p = *best; pick.push_back(p.i);
        vector<Act> keep; for (const Act& a : left) if (a.i != p.i && compat(a, p))
keep.push_back(a);
        left = keep;
    }
    sort(pick.begin(), pick.end());
    cout << "  shortest duration picks a2 (2 units) and then nothing:";
    for (int x : pick) cout << " a" << x;
    cout << "  -> size " << pick.size() << "\n";
    left = A; vector<int> pick2;
    while (!left.empty()) {
        const Act* best = &left[0];
        for (const Act& a : left) if (a.f < best->f) best = &a;
```

Listing 12.12 (Continued) *The programs for Exercises 12.1, 12.3, 12.5, 12.7, 12.9 and 12.11. Compile with g++ -std=c++17 -O2 -Wall -o exercises exercises.cpp and run ./exercises.*

```
        Act p = *best; pick2.push_back(p.i);
        vector<Act> keep; for (const Act& a : left) if (a.i != p.i && compat(a, p))
keep.push_back(a);
        left = keep;
    }
    sort(pick2.begin(), pick2.end());
    cout << "  earliest finish picks:";
    for (int x : pick2) cout << " a" << x;
    cout << "  -> size " << pick2.size() << "\n";
}

// ---------------------------------------------------------------- Ex 12.9
struct E { int a, b; long long w; };
static const string NM = "ABCDEFGH";
struct DSU {
    vector<int> p, r;
    explicit DSU(int n) : p(n), r(n,0) { iota(p.begin(), p.end(), 0); }
    int find(int x) { return p[x]==x ? x : p[x]=find(p[x]); }
    bool unite(int x, int y) {
        x = find(x); y = find(y);
        if (x == y) return false;
        if (r[x] < r[y]) swap(x, y);
        p[y] = x;
        if (r[x] == r[y]) ++r[x];
        return true;
    }
};
pair<long long,string> mstOf(vector<E> e, int n) {
    sort(e.begin(), e.end(), [](const E& x, const E& y){
        if (x.w != y.w) return x.w < y.w;
        if (x.a != y.a) return x.a < y.a;
        return x.b < y.b; });
    DSU d(n); long long t = 0; string s;
    for (const E& x : e) if (d.unite(x.a, x.b)) { t += x.w; s += string(1,NM[x.a]) +
string(1,NM[x.b]) + " "; }
    return {t, s};
}
void ex9() {
    cout << "\n=== Ex 12.9  transforming the weights ===\n";
    vector<E> G = { {0,1,3},{0,2,5},{1,2,1},{1,3,6},{2,3,4},{2,4,9},{3,4,2},
                    {3,5,8},{4,5,7},{4,6,10},{5,6,3},{5,7,12},{6,7,11} };
    auto base = mstOf(G, 8);
    cout << "  original       weight " << setw(5) << base.first << "   tree " <<
base.second << "\n";
    vector<E> plus = G; for (E& e : plus) e.w += 10;
    auto p = mstOf(plus, 8);
    cout << "  every w + 10   weight " << setw(5) << p.first << "   tree " << p.second
         << (p.second == base.second ? " (same tree)" : " (DIFFERENT)") << "\n";
    cout << "                 check: " << base.first << " + 7*10 = " << base.first + 70 <<
"\n";
```

Listing 12.12 (Continued) *The programs for Exercises 12.1, 12.3, 12.5, 12.7, 12.9 and 12.11. Compile with g++ -std=c++17 -O2 -Wall -o exercises exercises.cpp and run ./exercises.*

```
    vector<E> sq = G; for (E& e : sq) e.w = e.w * e.w;
    auto q = mstOf(sq, 8);
    cout << "  every w square weight " << setw(5) << q.first << "   tree " << q.second
         << (q.second == base.second ? " (same tree)" : " (DIFFERENT)") << "\n";
    vector<E> neg = G; for (E& e : neg) e.w = 20 - e.w;
    auto r = mstOf(neg, 8);
    cout << "  every 20 - w   weight " << setw(5) << r.first << "   tree " << r.second
         << (r.second == base.second ? " (same tree)" : " (DIFFERENT)") << "\n";
}

// ---------------------------------------------------------------- Ex 12.11
void ex11() {
    cout << "\n=== Ex 12.11  union by rank, measured ===\n";
    int n = 100000;
    {   // naive: attach the first root under the second, no compression
        vector<int> p(n); iota(p.begin(), p.end(), 0);
        function<int(int)> find = [&](int x){ while (p[x] != x) x = p[x]; return x; };
        for (int i = 1; i < n; ++i) { int a = find(i - 1), b = find(i); if (a != b) p[a] =
b; }
        long long steps = 0;
        for (int i = 0; i < n; ++i) { int x = i; while (p[x] != x) { x = p[x]; ++steps; } }
        cout << "  no rank, no compression:  total parent hops for " << n
             << " finds = " << steps << "\n";
    }
    {
        DSU d(n);
        for (int i = 1; i < n; ++i) d.unite(i - 1, i);
        long long steps = 0;
        for (int i = 0; i < n; ++i) { int x = i; while (d.p[x] != x) { x = d.p[x]; ++steps;
} }
        cout << "  union by rank + compression: total parent hops = " << steps << "\n";
    }
}

int main() { ex1(); ex3(); ex5(); ex7(); ex9(); ex11(); return 0; }
```

The output, which is where every number quoted in the solutions above came from:

```
=== Ex 12.1  greedy change-making with the coins {1, 4, 6} ===
  amount  8: greedy uses 3 coins, the optimum uses 2
  amount  9: greedy uses 4 coins, the optimum uses 3
  amount 14: greedy uses 4 coins, the optimum uses 3
  amount 15: greedy uses 5 coins, the optimum uses 4
  amounts 1..30 on which greed loses: 8

=== Ex 12.3  BFS from C instead of A ===
  source A   d: A=0 B=1 C=1 D=2 E=2 F=2 G=3 H=3 I=4
            pi: A<-- B<-A C<-A D<-B E<-B F<-C G<-D H<-E I<-G
  source C   d: A=1 B=2 C=0 D=3 E=1 F=1 G=2 H=2 I=3
            pi: A<-C B<-A C<-- D<-B E<-C F<-C G<-E H<-E I<-G
  the eccentricity of A is 4 and of C is 3, so the two trees differ in shape

=== Ex 12.5  the same digraph, two outer-loop orders ===
  outer loop A,B,C,D,E,F,G
    d/f: A=1/12 B=2/5 C=6/11 D=3/4 E=7/10 F=8/9 G=13/14
    tree 5, back 2, forward 1, cross 3
    A->B:T A->C:T A->D:F B->D:T C->D:C C->E:T D->A:B E->F:T F->E:B G->C:C G->F:C
  outer loop G,F,E,D,C,B,A
    d/f: A=4/7 B=5/6 C=2/13 D=3/8 E=9/12 F=10/11 G=1/14
    tree 6, back 4, forward 1, cross 0
    A->B:T A->C:B A->D:B B->D:B C->D:T C->E:T D->A:T E->F:T F->E:B G->C:T G->F:F
  the COUNT of back edges changed (2 then 4) but neither is zero:
  the verdict 'not acyclic' is order-independent, the classification is not

=== Ex 12.7  an instance where 'shortest duration first' fails ===
  a1=[0,5), a2=[4,6), a3=[5,10)
  shortest duration picks a2 (2 units) and then nothing: a2  -> size 1
  earliest finish picks: a1 a3  -> size 2

=== Ex 12.9  transforming the weights ===
  original       weight    31   tree BC DE AB FG CD EF GH
  every w + 10   weight   101   tree BC DE AB FG CD EF GH  (same tree)
                 check: 31 + 7*10 = 101
  every w square weight   209   tree BC DE AB FG CD EF GH  (same tree)
  every 20 - w   weight    79   tree FH GH EG CE DF BD AC  (DIFFERENT)

=== Ex 12.11  union by rank, measured ===
  no rank, no compression:  total parent hops for 100000 finds = 4999950000
  union by rank + compression: total parent hops = 99999
```

12.12.2 Where to read further

The definitive treatment of the material of Chapters 9 to 11, and so of this chapter, is Cormen, Leiserson, Rivest and Stein [1], whose Chapters 15, 19, 20 and 21 cover greedy algorithms, disjoint sets, elementary graph algorithms and minimum spanning trees respectively. The minimizing-maximum-lateness problem of Problem C is treated, with the same adjacent-exchange proof, by Kleinberg and Tardos [7]. The theory connecting matroids to the greedy algorithm, behind Problem F, is due to Edmonds [9]. The facts about canonical coin systems quoted in Problem A are from Kozen and Zaks [5] and Pearson [6]. Huffman's original 1952 paper [8] is four pages long and entirely readable. Kahn's topological-sort paper [4] is from 1962 and was motivated by exactly the scheduling problem of §12.2. Kruskal [2] and Prim [3] are the 1956 and 1957 originals; Tarjan [11] is where the $\alpha(n)$ bound of Problem V and Exercise 12.11 comes from. The counting problem of Problem P is #P-complete, as proved by Brightwell and Winkler [10]. For the discrete mathematics behind Cayley's formula and the summations used throughout, see Rosen [13].

References

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

[2] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, Feb. 1956.

[3] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, Nov. 1957.

[4] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962.

[5] D. Kozen and S. Zaks, "Optimal bounds for the change-making problem," *Theoretical Computer Science*, vol. 123, no. 2, pp. 377–388, 1994.

[6] D. Pearson, "A polynomial-time algorithm for the change-making problem," *Operations Research Letters*, vol. 33, no. 3, pp. 231–234, 2005.

[7] J. Kleinberg and É. Tardos, *Algorithm Design*. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[8] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.

[9] J. Edmonds, "Matroids and the greedy algorithm," *Mathematical Programming*, vol. 1, no. 1, pp. 127–136, 1971.

[10] G. Brightwell and P. Winkler, "Counting linear extensions," *Order*, vol. 8, no. 3, pp. 225–242, Sep. 1991.

[11] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, Apr. 1975.

[12] Republic of Indonesia, *Undang-Undang Republik Indonesia Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi* (Personal Data Protection Law), Jakarta, Indonesia, 2022.

[13] K. H. Rosen, *Discrete Mathematics and Its Applications*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2019.

Chapter 13
Single-Source Shortest Paths

By the end of this chapter you will be able to state the shortest-path problem precisely enough to argue about it, and to say what its answer is even in the cases where no answer exists. You will be able to prove that every subpath of a shortest path is itself a shortest path, and to prove the triangle inequality that every shortest-path algorithm secretly relies on. You will know exactly what a negative edge does to the problem (nothing) and what a negative cycle does to it (destroys it), and you will be able to tell the two apart in code. You will be able to write down the single primitive — *relaxation* — out of which every algorithm in this chapter is built, and prove the five properties that make it safe. You will be able to run Dijkstra's algorithm by hand on this chapter's example graph, printing the priority queue at every step, prove it correct, produce a graph on which it silently gives the wrong answer, and quote its running time for three different queue implementations. You will be able to do the same for Bellman–Ford: trace it pass by pass, prove that $|V| - 1$ passes are enough, and explain why one extra pass detects a negative cycle. You will be able to solve the problem in $\Theta(V + E)$ on a directed acyclic graph with no priority queue at all. And you will be able to recover the actual route, not merely its length, from the predecessor array — which is, after all, what anybody who asked the question actually wanted.

13.1 Where We Left Off

The standard reference treatment of this material is Chapter 22 of Cormen, Leiserson, Rivest and Stein [1], and the figures of this chapter follow its drawing conventions closely, so that a reader who later opens that book will feel at home. Two earlier chapters feed directly into this one, and it is worth saying exactly what each contributes.

13.1.1 What carries over from Chapters 10 and 11

From **Chapter 10** we take the graph and its vocabulary. A **directed graph** (or *digraph*) $G = (V, E)$ is a pair: a finite set V of **vertices** and a set $E \subseteq V \times V$ of **edges**, each edge being an *ordered* pair, so that (u, v) and (v, u) are different things. We write $|V|$ for the number of vertices and $|E|$ for the number of edges, and we keep this book's convention of dropping the bars inside asymptotic notation: $O(V + E)$ always means $O(|V| + |E|)$. A **path** from v_1 to v_k is a sequence of vertices $p = \langle v_1, v_2, \dots, v_k \rangle$ in which every consecutive pair is an edge, that is $(v_i, v_{i+1}) \in E$ for $i = 1,2, \dots, k - 1$. Its **length** is the number of edges, $k - 1$. A path is a **cycle** if $v_1 = v_k$ and $k > 1$. We store a graph in **adjacency lists**: $Adj[u]$ is the list of vertices u points to.

Chapter 10 also gave us breadth-first search, which computes, in $O(V + E)$ time, the fewest *number of edges* from a source s to every other vertex. That is the shortest-path problem for the special case where every edge counts the same. This chapter is what happens when edges stop counting the same.

From **Chapter 11** we take weights. A **weighted graph** is a graph together with a **weight function** $w: E \to \mathbb{R}$ that assigns a real number to each edge. Chapter 11 used weights to ask *what is the cheapest set of edges that keeps everything connected?* — the minimum-spanning-tree question — and answered it with Prim–Jarník and Kruskal. Chapter 11 also gave us the **priority queue**, with

its three operations `Insert`, `ExtractMin` and `DecreaseKey`, and the three implementations of it whose costs we shall need again here: an unsorted array, a binary heap, and a Fibonacci heap. And Chapter 11 gave us **topological sort**, which lists the vertices of a directed acyclic graph in an order that respects every arrow; §13.9 uses it directly.

> **DEFINITION (recalled from Chapter 11).** A **priority queue** on a set Q of elements, each holding a numeric *key*, supports: `Insert(Q, v)`, which adds v; `ExtractMin(Q)`, which removes and returns the element of smallest key; and `DecreaseKey(Q, v, k)`, which lowers v's key to k. The three implementations we use cost, per operation: unsorted array — $\Theta(V)$, $\Theta(1)$; binary heap — $O(\lg V)$, $O(\lg V)$; Fibonacci heap — $O(\lg V)$ amortized, $\Theta(1)$ amortized.

13.1.2 What is new

Chapter 11 measured a *whole network*: it found the cheapest way to keep everything joined up, and the answer was a set of edges with no particular starting point. This chapter measures a *journey*. There is a place you are starting from, there are places you want to reach, and the question is: what does it cost to get there, and by which route?

Those are not the same question, and it is worth being clear about the difference straight away, because students confuse them every year. A minimum spanning tree is the cheapest *connected subgraph*. A shortest-path tree is a tree of cheapest *routes from one particular vertex*. They are usually different trees, and neither one is the other's answer. Exercise 13.4 asks you to find a four-vertex graph where they differ, and Exercise 13.5 asks you to prove that a shortest-path tree can be strictly more expensive, in total edge weight, than the minimum spanning tree of the same graph.

13.2 The Shortest-Path Problem, Stated Precisely

13.2.1 From "number of edges" to "total weight"

Breadth-first search answers "how few *edges* away is it?". That is the right question when every edge is the same size — when the graph is a social network and an edge means *knows*, or a maze and an edge means *one step*. It is the wrong question the moment edges differ in size. A road between two districts of Surabaya may be two kilometers or twelve; a network link may cost one millisecond or forty; a courier leg may burn half a liter of petrol or three. When we attach such a number to every edge, the natural question changes from *how many edges* to *how much altogether*.

The generalization fits in one line: *generalize distance to the weighted setting*. Formally, we are handed a digraph $G = (V, E)$ together with a weight function $w: E \to \mathbb{R}$ assigning a real value to each edge. Note $\mathbb{R}$, not $\mathbb{R}^{\geq 0}$: the definition allows negative numbers, and §13.5 is about what that permission costs us.

13.2.2 The weight of a path

> **DEFINITION.** Let $p = \langle v_1, v_2, \ldots, v_k \rangle$ be a path in a weighted digraph. The **weight of the path** p, written $w(p)$, is the sum of the weights of its edges:

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}). \tag{13.1}$$

Read Equation (13.1) in words before you use it. The index i runs from 1 to $k-1$ — one value for each *edge* of the path, not for each vertex, which is why the upper limit is $k-1$ and not k. The term $w(v_i, v_{i+1})$ is the weight of the edge that leads from the i-th vertex of the path to the $(i+1)$-st. Adding them all up gives the total cost of walking the whole path from one end to the other. A path of one vertex, $p = \langle v_1 \rangle$, has no edges at all, so the sum is empty and $w(p) = 0$ — a convention we shall use constantly, because it is what makes $\delta(s, s) = 0$ come out right.

EXAMPLE. In the Dijkstra example graph of **Figure 13.6**, take the path $p = \langle s, x, u, v \rangle$. It has three edges. Their weights are $w(s, x) = 5$, $w(x, u) = 3$ and $w(u, v) = 1$. So

$$w(p) = 5 + 3 + 1 = 9.$$

Take instead $q = \langle s, u, v \rangle$, with two edges of weights $w(s, u) = 10$ and $w(u, v) = 1$, giving $w(q) = 11$. Both paths run from s to v; the first is cheaper even though it uses more edges. That is the whole point of weighting: *fewer edges* and *less cost* are different things.

13.2.3 The shortest-path weight

DEFINITION. The **shortest-path weight** from u to v, written $\delta(u, v)$ and read "delta of u, v", is

$$\delta(u, v) = \begin{cases} \min\{\, w(p) : p \text{ is a path from } u \text{ to } v \,\} & \text{if such a path exists,} \\ \infty & \text{otherwise.} \end{cases}$$

(13.2)

A **shortest path** from u to v is any path p from u to v with $w(p) = \delta(u, v)$.

Three points about definition Equation (13.2), each of which catches somebody out.

First, δ is a *number*, a shortest path is a *sequence of vertices*. They are different kinds of object and it is worth keeping separate words for them. When we say "compute the shortest paths" we usually mean "compute all the numbers $\delta(s, v)$, and enough extra bookkeeping to recover one path for each".

Second, the article is **a** shortest path, not **the**. There can be many, all of the same weight. In the Dijkstra example graph, $\delta(s, y) = 7$ by the unique route $\langle s, x, y \rangle$; but if we added an edge (s, y) of weight 7 there would be two shortest paths and no reason to prefer either. Algorithms return one of them and are not obliged to say which.

Third, $\delta(u, v) = \infty$ is not a failure, it is an answer: it means *there is no route at all*. In code we represent it by a large sentinel, and every comparison has to be written so that adding a weight to the sentinel does not overflow. All the programs in this chapter use the declaration `const long long INF = LLONG_MAX / 4;` — a quarter of the largest value a `long long` can hold, so that adding any realistic edge weight to it still fits — and guard every relaxation with `d[u] < INF`, which is exactly why they never wrap around.

13.2.4 A worked example, by hand

Let us compute $\delta(s, v)$ for the graph of **Figure 13.6** the slow, honest way: 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 $\langle s, x, u, v\rangle$. Enumerating paths is correct but hopeless in general: a graph on n vertices can have more than $(n-2)!$ simple paths between one pair.

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

So $\delta(s, v) = 9$. Why are there exactly six rows? A simple path may not visit a vertex twice, and 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 produces precisely these six routes. (A walk such as $s \to x \to u \to v \to y \to v$ is longer still, but it visits v twice, so it is not a simple path and cannot be shorter than the simple path it contains.) Look at what that cost us: six paths for a five-vertex graph. On thirty vertices the same method would list more paths than there are atoms in a grain of sand. The algorithms in this chapter get the same answer in time polynomial in $|V|$ and $|E|$ — Dijkstra in a few hundred machine operations for a graph this size — and the reason they can is the structural theorem of §13.3.

13.2.5 The four variants of the problem

The problem comes in four variants, and they are related in a way that is not symmetric.

> **DEFINITION.** Given a weighted digraph $G = (V, E, w)$: the **single-pair** problem asks for $\delta(u, v)$ and a shortest path, for one given pair u, v; the **single-source** problem asks for $\delta(s, v)$ and a shortest path, for one given source s and *every* $v \in V$; the **single-destination** problem asks for $\delta(u, t)$ for one given destination t and every $u \in V$; the **all-pairs** problem asks for $\delta(u, v)$ for every ordered pair $u, v \in V$.

Table 13.2 sets the four side by side, with the reduction each one uses.

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

13.2.6 Why single-source is the one we solve

The single-pair problem looks easier than the single-source problem — surely computing one number is less work than computing $|V|$ of them? — and the surprise is that nobody knows how to exploit that. Every algorithm we have for the single-pair problem is a single-source algorithm that happens to be allowed to stop as soon as the target leaves the priority queue. In the worst case the target is the last vertex to leave, and you have solved the whole single-source problem

anyway. Put briefly: *a solution to the single-source problem solves the single-pair problem efficiently, too*. There is no asymptotically better method known for the general case, and this is one of the oldest open questions in the subject.

The single-destination problem is the single-source problem in a mirror. Build the **transpose graph** $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$ where $E^{\mathrm{T}} = \{(v, u) : (u, v) \in E\}$, keeping each edge's weight, and run a single-source algorithm from t in G^{T}. A path $u \rightsquigarrow t$ in G is exactly a path $t \rightsquigarrow u$ in G^{T} with the same weight, so the numbers come out identical. Reversing the edges costs $\Theta(V + E)$, which is free compared with anything else we do.

The all-pairs problem can be solved by running a single-source algorithm once from every vertex — $|V|$ runs. For a sparse graph with non-negative weights that is actually competitive: $|V|$ runs of Dijkstra with a Fibonacci heap costs $O(V^2 \lg V + VE)$, which beats the $\Theta(V^3)$ of Floyd–Warshall when E is small. Chapter 15 takes this up properly, including Johnson's algorithm, whose whole idea is to *reweight* a graph with negative edges so that Dijkstra may legally be run on it.

So: three of the four variants either reduce to single-source or are built out of it. That is why this chapter solves single-source and nothing else.

13.2.7 Where the problem actually turns up

Three classical applications are worth a sentence each, because each stresses the problem differently.

The general theory behind all of them — shortest paths as the simplest case of a minimum-cost network flow — is the subject of Ahuja, Magnanti and Orlin [2], and a gentler algorithmic treatment of the same ground is Kleinberg and Tardos [3].

Static and dynamic network routing. Every packet that crosses the internet is routed by an algorithm descended from the two in this chapter. The link-state protocol OSPF has every router build a map of the whole network and run Dijkstra's algorithm on it. The distance-vector protocol RIP has every router keep only a table of distances and exchange it with its neighbors — which is Bellman–Ford, run in a distributed fashion, one pass per exchange round. The two families of protocol are, quite literally, the two algorithms of this chapter with a network stack around them.

Robot motion planning. Discretize the floor into a grid or a graph of landmarks, weight each edge by the time or energy the robot needs to traverse it, and the cheapest safe route is a shortest path. Weights here are naturally non-negative, so Dijkstra applies.

Map and route generation in traffic. This is the application everyone has in their pocket. Weights are travel times, which change through the day, so the graph is re-weighted continuously and the query is single-pair. Production systems use heavily engineered variants — A* with a geometric heuristic, contraction hierarchies — but the correctness of all of them is argued from exactly the properties proved in §13.6.

To that list add two more that this book returns to. **Scheduling with precedences:** if tasks form a DAG and edge weights are durations, the *longest* path is the critical path of the project — §13.9.7 shows that this is the same algorithm with the sign flipped. And **currency arbitrage:** take exchange rates, weight each edge by $-\ln(\text{rate})$, and a negative cycle is precisely a sequence of trades that returns more money than it started with. Bellman–Ford's negative-cycle detector is an arbitrage detector.

13.3 Optimal Substructure of Shortest Paths

Everything in this chapter rests on one structural fact, and it is worth stating it before any algorithm, because every algorithm is an exploitation of it.

13.3.1 The theorem

THEOREM 13.1 (Optimal substructure of shortest paths). Subpaths of shortest paths are shortest paths. Precisely: let $p = \langle v_1, v_2, \dots, v_k \rangle$ be a shortest path from v_1 to v_k, and for any $1 \le i \le j \le k$ let $p_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ be the subpath of p from v_i to v_j. Then p_{ij} is a shortest path from v_i to v_j.

Before the proof, be clear what is being claimed, because the statement is easy to nod at and hard to use. It says: if the cheapest way from Depot to Keputih happens to pass through ITS, then the part of that journey from Depot to ITS is itself the cheapest way from Depot to ITS, and the part from ITS to Keputih is itself the cheapest way from ITS to Keputih. Not *a* cheap way — *the* cheapest. You cannot build an optimal whole out of a sub-optimal part.

13.3.2 The proof, by cutting and pasting

The whole proof fits in one line: *if some subpath were not a shortest path, one could substitute the shorter subpath and create a shorter total path*. Here it is at full length, one step at a time.

Proof. Decompose the path p into three consecutive pieces at the two chosen vertices:

$$v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k.$$

Every edge of p lies in exactly one of the three pieces, so the weights add:

$$w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk}). \tag{13.3}$$

Now suppose, for the sake of contradiction, that p_{ij} is *not* a shortest path from v_i to v_j. Then by the definition of δ there is some path p'_{ij} from v_i to v_j with

$$w(p'_{ij}) < w(p_{ij}). \tag{13.4}$$

Build a new path p' from v_1 to v_k by *cutting out* p_{ij} and *pasting in* p'_{ij}:

$$v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k.$$

This is a legitimate path: p_{1i} ends at v_i and p'_{ij} begins at v_i, so they join; p'_{ij} ends at v_j and p_{jk} begins at v_j, so they join too. Its weight is

$$\begin{aligned} w(p') = w(p_{1i}) + w(p'_{ij}) + w(p_{jk}) \;&<\; w(p_{1i}) + w(p_{ij}) + w(p_{jk}) \\ &=\; w(p), \end{aligned}$$

where the strict inequality comes from Equation (13.4) and the last equality is Equation (13.3). So $w(p') < w(p)$: we have produced a path from v_1 to v_k strictly cheaper than p. But p was assumed to be a *shortest* path from v_1 to v_k, so no such path exists. Contradiction. Therefore no cheaper p'_{ij} exists, and p_{ij} is a shortest path from v_i to v_j. ■

13.3.3 Reading the proof again, slowly

The argument is called **cut and paste** and you met it in Chapter 9 for greedy algorithms and in Chapter 7 for dynamic programming. It always has the same four moves, and it is worth naming them, because you will need to reproduce it yourself, and Chapter 16 works problems that call for exactly this argument.

1. **Assume the sub-solution is not optimal.** Here: assume p_{ij} is beatable.
2. **Take the better sub-solution that the assumption hands you.** Here: p'_{ij} with $w(p'_{ij}) < w(p_{ij})$.
3. **Splice it into the whole.** Here: replace the middle section and check the result is still a legal path — this step is where the argument can fail for other problems, so never skip the check.
4. **Observe that the whole has improved, contradicting its optimality.**

Step 3 is the one that deserves suspicion. It works here because a path is just a sequence of edges and any two paths that share an endpoint can be concatenated. It does *not* work for, say, longest **simple** paths: splicing in a cheaper subpath can revisit a vertex and make the result not simple. That is exactly why the longest-simple-path problem is NP-hard while the shortest-path problem is easy, and it is a distinction worth carrying.

13.3.4 What optimal substructure buys us

Optimal substructure is the license to build shortest paths *incrementally*. If I know a shortest path from s to u, and (u, v) is an edge, then the best I can do for v *through* u is $\delta(s, u) + w(u, v)$ — I do not need to reconsider the route to u, because the theorem guarantees the prefix of an optimal route to v through u is an optimal route to u. Every algorithm in this chapter is an answer to the question *in what order should I consider the vertices, so that when I use* $\delta(s, u)$ *it is already known?* Dijkstra answers *in increasing order of distance*; Bellman–Ford answers *I don't know, so try everything* $|V| - 1$ *times*; the DAG algorithm answers *in topological order*. That is the entire chapter in three sentences.

13.4 The Triangle Inequality

13.4.1 Statement

THEOREM 13.2 (Triangle inequality, vertex form). For any three vertices u, x, v of a weighted digraph,

$$\delta(u, v) \leq \delta(u, x) + \delta(x, v). \tag{13.5}$$

The name of Equation (13.5) comes from geometry: the straight side of a triangle is never longer than the other two sides together. The graph version says the same thing about routes. Going from u to v directly (that is, by the best available route) is never worse than being forced to detour through x.

13.4.2 The proof

Figure 13.1 draws the situation: a wavy arrow straight from u to v, and a second route that dips down through x. The wavy lines are the standard notation for "some path, we do not care which".

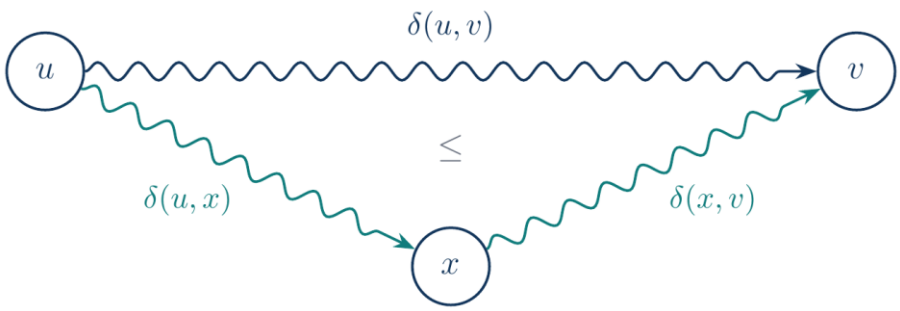


Figure 13.1 The triangle inequality: the upper wavy arrow is a shortest path from u to v, the lower pair a shortest path from u to x followed by one from x to v. The lower route is *a* path from u to v, so it cannot be cheaper than the best one.

Proof. There are three cases, and the degenerate ones matter.

Case 1: either $\delta(u,x) = \infty$ or $\delta(x,v) = \infty$. Then the right-hand side is ∞ and the inequality holds trivially, whatever $\delta(u,v)$ is. (This is why we define $\infty + c = \infty$.)

Case 2: both are finite, and $\delta(u,v)$ is finite. Let p_1 be a shortest path from u to x, so $w(p_1) = \delta(u,x)$, and let p_2 be a shortest path from x to v, so $w(p_2) = \delta(x,v)$. Concatenating them gives a path $q = p_1 p_2$ from u to v — legal, because p_1 ends where p_2 begins — of weight

$$w(q) = w(p_1) + w(p_2) = \delta(u,x) + \delta(x,v).$$

Now $\delta(u,v)$ is by definition the *minimum* weight over all paths from u to v, and q is one such path. A minimum over a set is at most any particular member of that set, so

$$\delta(u,v) \;\leq\; w(q) \;=\; \delta(u,x) + \delta(x,v).$$

Case 3: both terms on the right are finite but $\delta(u,v) = \infty$. This cannot happen: Case 2 exhibited an actual path q from u to v, so the set of paths from u to v is non-empty and its minimum is finite. ■

The one-line version — *the shortest path $u \rightsquigarrow v$ is no longer than any other path $u \rightsquigarrow v$, in particular the one concatenating the shortest $u \rightsquigarrow x$ with the shortest $x \rightsquigarrow v$* — is exactly Case 2, and it is the sentence to remember.

13.4.3 The edge form, which is the one we use

For algorithms, the useful special case is the one where the detour is a single edge.

COROLLARY 13.3 (Triangle inequality, edge form). For any edge $(u,v) \in E$ and any source s,

$$\delta(s,v) \;\leq\; \delta(s,u) + w(u,v).$$

(13.6)

Proof. Apply Theorem 13.2 with the triple (s,u,v): $\delta(s,v) \leq \delta(s,u) + \delta(u,v)$. Then note that the single edge $\langle u,v\rangle$ is a path from u to v of weight $w(u,v)$, so $\delta(u,v) \leq w(u,v)$ by the definition of δ as a minimum. Substituting gives the result. ■

Stare at Equation (13.6) until it feels obvious, because the whole of §13.6 is built on it. It says: *the true distance to v can never be beaten by going to u optimally and then taking one more edge.* Everything the algorithms do is try to make their estimate $d[v]$ satisfy this with equality.

13.4.4 A worked check

Take the Dijkstra example graph again, whose true distances from s we computed as $\delta(s,s) = 0$, $\delta(s,x) = 5$, $\delta(s,y) = 7$, $\delta(s,u) = 8$, $\delta(s,v) = 9$. Check the edge form on all ten edges, as **Table 13.3** does:

Table 13.3 The edge form of the triangle inequality checked on every edge of the Dijkstra example graph. The rows showing *equality* — (s,x), (x,u), (u,v), (x,y) — are precisely the four edges of the shortest-path tree, as §13.6.9 explains.

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

13.5 Negative Weights and Negative Cycles

The definition of §13.2 allowed w to take negative values. This section is about what that permission costs. The distinction it draws — between a negative *edge*, which is harmless, and a negative *cycle*, which is fatal — is the single most misunderstood point in the whole topic.

13.5.1 A negative edge is not a problem

Negative weights are not exotic. A courier leg that ends at a partner drop-point which pays a rebate has a negative net cost. A chemical reaction that releases energy has a negative energy change. A trading step that earns more than it spends has negative cost. In each case, the arithmetic of §13.2 goes through untouched: $w(p)$ is still a sum, $\delta(u,v)$ is still a minimum, and Theorems 13.1 and 13.2 were proved without ever assuming $w \ge 0$. Go back and check: the cut-and-paste proof only ever *added* and *compared* weights. Nothing there needs them to be positive.

The Bellman–Ford example graph of §13.8 has three negative edges — $w(x,t) = -2$, $w(y,x) = -3$ and $w(t,z) = -4$ — and a perfectly well-defined answer, which §13.8 computes.

13.5.2 A negative cycle is fatal

Now suppose the graph contains a cycle c whose total weight is negative, $w(c) < 0$, and suppose c is reachable from the source s — that is, there is some path from s to a vertex on c.

> **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. Let c be the cycle, let x be any vertex on it, let p be a path $s \rightsquigarrow x$, and let q be a path $x \rightsquigarrow v$. Then for every integer $k \ge 0$ the walk

$$s \overset{p}{\rightsquigarrow} x \underbrace{\overset{c}{\rightsquigarrow} x \overset{c}{\rightsquigarrow} x \cdots \overset{c}{\rightsquigarrow} x}_{k \text{ laps}} \overset{q}{\rightsquigarrow} v$$

has weight $w(p) + k\,w(c) + w(q)$. Since $w(c) < 0$, this decreases without bound as k grows: choose k large enough and the "length" is below any number you name. There is no minimum, so $\delta(s, v)$ is undefined, and $-\infty$ is the conventional way of recording that.

Figure 13.2 shows the smallest interesting instance. The cycle $a \to b \to c \to a$ has weight $2 + (-5) + 1 = -2$. It is reachable from s by the edge (s, a), and d is reachable from it by the edge (c, d). So $\delta(s, a) = \delta(s, b) = \delta(s, c) = \delta(s, d) = -\infty$, while $\delta(s, s) = 0$ — the source itself is safe only because no edge leads back into s here.

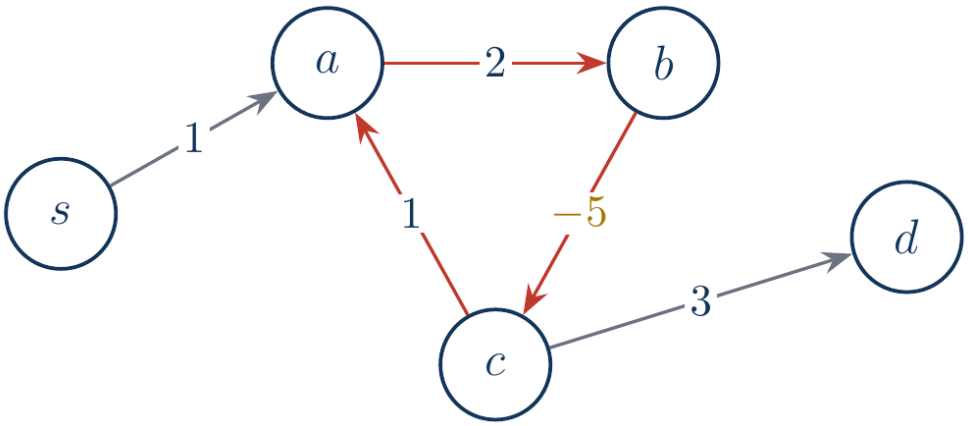


Figure 13.2 A graph with a reachable negative cycle. Going round the red cycle $a \to b \to c \to a$ costs $2 + (-5) + 1 = -2$, so each extra lap makes the "path" two cheaper. Every vertex reachable from the cycle — here a, b, c and d — has $\delta(s, \cdot) = -\infty$. The problem is not hard on this graph; it is *ill-posed*.

Note carefully what is *not* claimed. A negative cycle that is **unreachable** from s does no harm at all: you cannot get onto it, so it cannot corrupt any distance from s. And a vertex not reachable *from* the cycle keeps its ordinary finite distance. Bellman–Ford as written in **Figure 13.13** returns `false` as soon as it sees any reachable negative cycle, which is the safe and usual behavior; Exercise 13.13 asks you to modify it so that it labels each vertex $+\infty$, a finite number, or $-\infty$ individually, and the solution's program does exactly that.

13.5.3 Shortest paths have no cycles

A second observation about negative weights is a theorem in disguise.

> **THEOREM 13.4.** If the graph contains no negative-weight cycle reachable from s, then for every vertex v reachable from s there is a shortest path from s to v that is **simple** — it repeats no vertex — and therefore has at most $|V| - 1$ edges.

Proof. Let p be any shortest path from s to v; by hypothesis $\delta(s, v)$ is finite, so one exists. Suppose p repeats a vertex x. Then p has the shape

$$s \rightsquigarrow x \overset{c}{\rightsquigarrow} x \rightsquigarrow v$$

where c is the cycle between the two visits to x. Removing c leaves a path p' from s to v of weight $w(p') = w(p) - w(c)$.

Now, c is reachable from s (we reached it along p), so by hypothesis $w(c) \geq 0$. Hence $w(p') = w(p) - w(c) \leq w(p)$. But p was a shortest path, so $w(p') \geq \delta(s, v) = w(p)$. The two inequalities

force $w(p') = w(p)$: the shorter path is *also* a shortest path. Repeat the excision for every repeated vertex; each step removes at least one edge, so the process terminates, and what remains is a simple shortest path.

A simple path visits each vertex at most once, so it has at most $|V|$ vertices and hence at most $|V| - 1$ edges. ■

Stated as a slogan: *any shortest path in graph G can be no longer than $n - 1$ edges, where n is the number of vertices.* Write that on the inside of your eyelids. It is the reason Bellman–Ford runs exactly $|V| - 1$ passes — not $|V|$, not $|V| + 1$ — and it is the reason the $|V|$-th pass is a negative-cycle detector rather than useful work.

> **EXAMPLE.** The Dijkstra example graph has five vertices, so every shortest path from s has at most four edges. Check: the longest one we found was $\langle s, x, u, v\rangle$, three edges. The Bellman–Ford example graph also has five vertices, and its longest shortest path is $\langle s, y, x, t, z\rangle$ — exactly four edges, the maximum allowed. That is why the Bellman–Ford example of §13.8.4 needs all four passes and would be wrong with three.

13.5.4 Which algorithm survives what

We have not met the algorithms yet, but it is worth having **Table 13.4** in view from the start, because the row that matters is the second one.

Table 13.4 What each of the three algorithms does in each weight regime. The word to notice is *silently*: Dijkstra does not crash or complain on a negative edge, it returns a plausible-looking wrong number. §13.7.3 builds the smallest example.

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

13.6 Relaxation: the One Primitive

Every algorithm in this chapter — and every algorithm in Chapter 15 too — is built from a single operation. Learn it once here and the rest of the chapter is bookkeeping about *when* to apply it.

13.6.1 The two arrays: d and π

> **DEFINITION.** For each vertex v we maintain two values. $d[v]$, the **shortest-path estimate**, is a number that is always an *upper bound* on $\delta(s, v)$: the weight of the best route from s to v that we happen to have found so far. It starts at ∞ for every vertex except s, where it starts at 0. $\pi[v]$, the **predecessor**, is the vertex from which we arrived at v along that best-so-far route, or `NIL` if we have not found any route to v yet.

The pair (d, π) is what every one of our algorithms is computing; the numbers are the answer to *how far*, the predecessors are the answer to *which way*, and §13.10 shows how to turn the second into an actual list of vertices.

WARNING. $d[v]$ is an *estimate*, and during the run it can be too large. It is never too small — that is the upper-bound property of §13.6.5, and it is the first thing we prove. Beginners write code that treats $d[v]$ as the answer halfway through the loop; that is the commonest bug in this whole subject.

13.6.2 Initialization

The first lines of all three algorithms are identical, and the textbooks give them a name, **Pseudocode 13.1**:

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
```

This is lines 1–3 of Dijkstra's pseudocode in **Figure 13.4**, lines 01–04 of Bellman–Ford in **Figure 13.13**, and lines 01–03 of the DAG algorithm in **Figure 13.19**. It costs $\Theta(V)$ and it says: *I know one route, the empty route from s to itself, and it costs nothing; about everything else I know nothing at all.*

13.6.3 The relaxation step

DEFINITION. To **relax** an edge (u, v) is to test whether the best route to v found so far can be improved by going to u and then taking the edge (u, v), and, if so, to record the improvement. **Pseudocode 13.2** states it exactly.

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
```

That is all of it: one comparison, and two assignments if the comparison succeeds. These same three lines appear, highlighted, in Dijkstra's pseudocode of **Figure 13.4** under the label *relaxing edges*, because they are the only lines of Dijkstra that do anything to the answer.

The word "relax" is borrowed from the physical analogy of a stretched spring being allowed to settle. The estimate $d[v]$ starts stretched taut at ∞ and every successful relaxation lets it shrink a little closer to its resting value $\delta(s, v)$. It only ever goes down, never up — look at line 1: the assignment happens only when the new value is *smaller*.

13.6.4 A worked relaxation

Figure 13.3 shows the two outcomes on the same tiny picture. Each node is labeled with its name on top and its current d value underneath, and the edge (u, v) has weight 2.

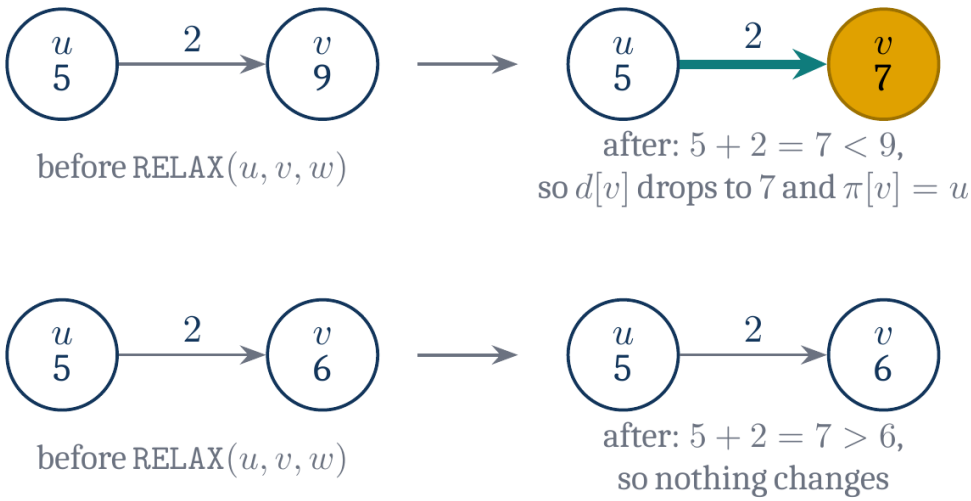


Figure 13.3 The two outcomes of `RELAX(u,v,w)` with $w(u,v) = 2$. **Top:** $d[u] + w = 5 + 2 = 7$, which is less than the current $d[v] = 9$, so $d[v]$ drops to 7 and $\pi[v]$ becomes u. **Bottom:** $d[u] + w = 7$, which is *not* less than the current $d[v] = 6$, so nothing at all happens. A relaxation that changes nothing is not a wasted step; it is a confirmation that the current estimate already beats this route.

In the top case: we knew a route to v costing 9. We know a route to u costing 5. Adding one edge of weight 2 gives a route to v costing 7, which is better, so we take it and remember that we arrived via u. In the bottom case: the route through u would cost 7, and we already have one costing 6, so we keep what we have and leave $\pi[v]$ alone.

> **KEY IDEA.** Relaxation is the *only* way d and π ever change after initialization. Dijkstra, Bellman–Ford and the DAG algorithm differ only in **the order in which they relax edges, and how many times**. Everything we prove about d and π in this section therefore applies to all three at once.

13.6.5 The upper-bound property

> **LEMMA 13.5 (Upper-bound property).** After `INITIALIZE-SINGLE-SOURCE`, and after any sequence of relaxation steps whatever, $d[v] \geq \delta(s,v)$ holds for every $v \in V$. Moreover, once $d[v]$ reaches the value $\delta(s,v)$ it never changes again.

Proof, by induction on the number of relaxation steps performed.

Base case: zero steps. Immediately after initialization, $d[s] = 0$ and $d[v] = \infty$ for $v \neq s$. For $v \neq s$, $\infty \geq \delta(s,v)$ trivially. For s itself we need $0 \geq \delta(s,s)$. Now $\delta(s,s) \leq 0$, because the one-vertex path $\langle s \rangle$ has weight 0; and since we have assumed no negative cycle is reachable from s, no cycle through s has negative weight, so no route from s back to s costs less than 0, giving $\delta(s,s) = 0$. Hence $d[s] = 0 = \delta(s,s)$. The base case holds.

Inductive step. Assume $d[x] \geq \delta(s,x)$ for every x, and let the next step be `RELAX(u,v,w)`. Only $d[v]$ can change, and only to the value $d[u] + w(u,v)$. So it suffices to show

$$d[u] + w(u,v) \;\geq\; \delta(s,v).$$

By the inductive hypothesis $d[u] \geq \delta(s,u)$, so

$$d[u] + w(u,v) \;\geq\; \delta(s,u) + w(u,v) \;\geq\; \delta(s,v),$$

the second inequality being exactly Corollary 13.3, the edge form of the triangle inequality. Therefore the new $d[v]$, whatever it is, is still at least $\delta(s, v)$. ■

The second sentence of the lemma. Suppose $d[v] = \delta(s, v)$ at some moment. Any later change to $d[v]$ would come from a successful relaxation, which by line 1 of `RELAX` strictly *decreases* $d[v]$, giving $d[v] < \delta(s, v)$ — impossible by the part just proved. So $d[v]$ is frozen. ■

In plain words: **the estimate is never optimistic, and once it is right it stays right.** That sentence is the license for every algorithm here to stop as soon as no relaxation changes anything.

13.6.6 The no-path property

> **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.

Proof. That $\delta(s, v) = \infty$ is the definition. By Lemma 13.5, $d[v] \geq \delta(s, v) = \infty$, and since ∞ is the largest value our arithmetic carries, $d[v] = \infty$. ■

It is worth seeing *why* $d[v]$ can never drop. For $d[v]$ to become finite, some relaxation of an edge (u, v) must have found $d[u]$ finite. Chase that back: $d[u]$ became finite because of a relaxation of some (t, u) with $d[t]$ finite, and so on. The chain must terminate at s, the only vertex whose d was finite at the start — and that chain is a path from s to v, contradicting the assumption that none exists.

This is the property that lets us write `if (d[u] < INF && d[u] + e.w < d[e.v])` in every program in this chapter and be sure we are not silently doing arithmetic on a sentinel.

13.6.7 The convergence property

> **LEMMA 13.7 (Convergence property).** Suppose $s \rightsquigarrow u \to v$ is a shortest path from s to v for some vertices u, v — that is, $(u, v) \in E$ and $\delta(s, v) = \delta(s, u) + w(u, v)$. If at some moment $d[u] = \delta(s, u)$, then at any time *after* the edge (u, v) is relaxed, $d[v] = \delta(s, v)$.

Proof. Consider the moment just after the relaxation of (u, v). Two things could have happened before it.

If $d[v]$ was already equal to $\delta(s, v)$, then by the freezing half of Lemma 13.5 it is still equal to $\delta(s, v)$, and we are done.

Otherwise $d[v] > \delta(s, v)$ just before the relaxation. At the moment of relaxation, $d[u] = \delta(s, u)$ — it reached that value earlier and, by Lemma 13.5 again, cannot have moved since. So the test on line 1 of `RELAX` compares

$$d[v] \;>\; \delta(s, v) \;=\; \delta(s, u) + w(u, v) \;=\; d[u] + w(u, v),$$

which is true, so line 2 fires and sets $d[v] \leftarrow d[u] + w(u, v) = \delta(s, v)$. Afterward, Lemma 13.5 freezes it. ■

In plain words: **if you have the right answer for u, and u is the last stop before v on a best route, then relaxing that one edge gives you the right answer for v, permanently.** This is the workhorse lemma. Every correctness proof in this chapter is an argument that the algorithm relaxes the edges of some shortest path in an order that lets Lemma 13.7 fire repeatedly.

13.6.8 The path-relaxation property

LEMMA 13.8 (Path-relaxation property). Let $p = \langle v_0, v_1, \dots, v_k \rangle$ be a shortest path from $s = v_0$ to v_k. Suppose the edges of p are relaxed **in order** — (v_0, v_1), then (v_1, v_2), …, then (v_{k-1}, v_k) — possibly with any number of other relaxations interleaved anywhere. Then after the last of these k relaxations, $d[v_k] = \delta(s, v_k)$, and it stays so for ever.

Proof, by induction on i, the claim being: after the edge (v_{i-1}, v_i) has been relaxed, $d[v_i] = \delta(s, v_i)$.

Base case $i = 0$. Before any relaxation at all, $d[v_0] = d[s] = 0 = \delta(s, s)$ as shown in Lemma 13.5. So the claim holds vacuously for the start of the path.

Inductive step. Assume $d[v_{i-1}] = \delta(s, v_{i-1})$ holds at the moment the edge (v_{i-1}, v_i) is relaxed. (By Lemma 13.5 it held from some earlier moment onwards and cannot have been disturbed.) Since p is a shortest path from s to v_k, Theorem 13.1 says its prefix $\langle v_0, \dots, v_i \rangle$ is a shortest path from s to v_i, and hence $\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i)$. That is precisely the hypothesis of Lemma 13.7 with $u = v_{i-1}$, $v = v_i$. Applying it, after the relaxation $d[v_i] = \delta(s, v_i)$, permanently.

Taking $i = k$ gives the lemma. ■

Two things to notice, because both are used later and both surprise people.

The interleaving is free. Other relaxations may happen between the relaxations of p's edges, in any number and any order. They cannot hurt: by Lemma 13.5 they can only lower estimates, never below δ, so the hypothesis $d[v_{i-1}] = \delta(s, v_{i-1})$ of each step survives. This is what makes Bellman–Ford work: it relaxes *every* edge in every pass, so in particular it relaxes the edges of p in the right order, with a great deal of irrelevant work mixed in.

Only the relative order matters. We need (v_0, v_1) relaxed before (v_1, v_2) before (v_2, v_3) and so on. We do not need them consecutive, and we do not need to know what p is. A pass over all edges guarantees at least one more step of the chain, which is exactly the counting argument of §13.8.3.

13.6.9 The predecessor-subgraph property

We have been maintaining π and saying nothing about it. Here is what it is worth.

DEFINITION. The **predecessor subgraph** $G_\pi = (V_\pi, E_\pi)$ is defined by

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}, \qquad E_\pi = \{(\pi[v], v) : v \in V_\pi \setminus \{s\}\}.$$

(13.7)

In words, Equation (13.7) says: keep every vertex that has a predecessor (plus s itself), and draw one edge into each such vertex, from its predecessor.

DEFINITION. A **shortest-path tree** rooted at s is a directed subgraph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$ such that (i) V' is exactly the set of vertices reachable from s; (ii) G' is a rooted tree with root s; and (iii) for every $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

LEMMA 13.9 (Predecessor-subgraph property). If the graph contains no negative-weight cycle reachable from s, and if at some moment $d[v] = \delta(s, v)$ for every $v \in V$, then G_π is a shortest-path tree rooted at s.

Proof sketch, in three steps. *It is acyclic.* Suppose G_π contained a cycle $\langle c_0, c_1, \dots, c_k = c_0 \rangle$, with $\pi[c_i] = c_{i-1}$. Each $\pi[c_i]$ was set by a relaxation of (c_{i-1}, c_i) at a moment when $d[c_i]$ became

$d[c_{i-1}] + w(c_{i-1}, c_i)$; afterward $d[c_{i-1}]$ can only have decreased, so at the present moment $d[c_i] \geq d[c_{i-1}] + w(c_{i-1}, c_i)$. Summing round the cycle, every d-term appears once on each side and cancels, leaving $0 \geq w(\text{cycle})$: the cycle's weight is not positive. If it were negative we would contradict the hypothesis; the remaining case, weight exactly zero, is ruled out in Exercise 13.10 by looking at the moment the cycle was closed. Since $\pi[s]$ is never set, s has in-degree 0, and every other vertex of V_π has in-degree exactly 1 (it has exactly one π value). An acyclic digraph in which one vertex has in-degree 0 and all others in-degree 1 is a rooted tree.

It spans the right set. $\pi[v] \neq \text{NIL}$ exactly when $d[v]$ became finite, which by Lemma 13.6 happens exactly when v is reachable from s.

Its paths are shortest paths. Walk back from v along π: $v, \pi[v], \pi[\pi[v]], \ldots$ until reaching s. At the moment $\pi[v]$ was last set we had $d[v] = d[\pi[v]] + w(\pi[v], v)$, and since no d has changed since (all are at their final values δ), $\delta(s, v) = \delta(s, \pi[v]) + w(\pi[v], v)$. Applying that at each step and summing telescopes to: the weight of the π-path from s to v equals $\delta(s, v)$. ■

This lemma is the license for §13.10: it says you may print a shortest path by following π backward, and what you get really is a shortest path.

13.6.10 The five properties on one page

Table 13.5 collects them, with what each one is for.

Table 13.5 The five properties of relaxation. They are proved once, here, and used by all three algorithms; no algorithm in this chapter needs a correctness argument that does not reduce to this table.

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

13.7 Dijkstra's Algorithm

13.7.1 The idea

Edsger Dijkstra's algorithm [4], published in 1959 and reportedly designed in about twenty minutes in a café in Amsterdam, solves the single-source problem for graphs whose edge weights are all **non-negative**. Its character can be summed up in five short statements, and each is worth unpacking.

Non-negative edge weights. The restriction is real and §13.7.3 shows exactly where it bites.

Greedy, similar to Prim's algorithm for MST. Chapter 11's Prim grows one tree outward from a root, always adding the cheapest edge that leaves the tree. Dijkstra grows one tree outward from the source, always adding the vertex whose *total distance from the source* is smallest. The two algorithms have the same shape; only the key differs. Prim's key for v is $\min\{w(u, v) : u \in \text{tree}\}$,

the cost of *one edge*; Dijkstra's key for v is $\min\{d[u] + w(u, v) : u \in S\}$, the cost of *the whole journey*. Get that difference straight and you will never confuse the two again.

Like breadth-first search — if all weights equal 1*, one can simply use BFS.* With unit weights, "closest vertex not yet finished" is "fewest edges away", and the priority queue degenerates into the FIFO queue of Chapter 10.

Use Q, a priority queue keyed by $d[v]$. BFS used a first-in-first-out queue because all frontier vertices were equally good. Here they are not, so we need a structure that can hand back the smallest, and that can be re-ordered when a d value drops.

Basic idea: maintain a set S of solved vertices; at each step select the "closest" vertex u, add it to S, and relax all edges out of u.

> **KEY IDEA.** Dijkstra processes vertices in increasing order of their true distance from s. Once a vertex leaves the priority queue, its d value is *final* and is never revised. The entire correctness proof is the claim that this is safe — and it is safe only because no edge weight is negative, so leaving S can never be made cheaper by wandering further out first.

13.7.2 The pseudocode

Figure 13.4 gives the pseudocode, with the relaxation step highlighted.

```
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.

Line by line, in words:

Line 1–2. For every vertex v, set its estimate $d[v]$ to ∞. We know nothing yet. **Line 3.** Set $d[s] \leftarrow 0$. The source is zero away from itself. **Line 4.** $S \leftarrow \emptyset$. S is the set of vertices whose answer is final. The comment in the pseudocode calls them the *solved* vertices, the clearest word for them. **Line 5.**

$Q \leftarrow V$. Put every vertex into the priority queue, keyed by its current d. At this moment s has key 0 and everything else has key ∞. **Line 6.** While the queue is not empty — so the loop body runs exactly $|V|$ times. **Line 7.** $u \leftarrow \text{Extract} - \text{Min}(Q)$. Take out the vertex with the smallest estimate. First time round this must be s, since everything else is at ∞. **Line 8.** $S \leftarrow S \cup \{u\}$. Declare u solved. From now on $d[u]$ will never change. **Line 9.** For every vertex v that u points to… **Lines 10–11.** …relax the edge (u, v): if the route through u beats the best route to v known so far, take it. One implementation note matters here: *setting* $d[v]$ *updates* Q *(needs* `Decrease-Key`*)*. The priority queue is keyed by d, so lowering $d[v]$ must lower v's key in the queue, or the queue will hand back the wrong vertex next time.

Notice what the pseudocode does *not* do: it never sets π. The figure leaves π implicit; it appears explicitly in **Pseudocode 13.2** and in the Bellman–Ford pseudocode of §13.8.2. Every implementation in this chapter sets $\pi[v] \leftarrow u$ alongside line 11, because without it you get distances and no routes; see §13.10.

13.7.3 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 every $w \geq 0$, that is true, and so the vertex with the smallest estimate cannot possibly be improved by a route through any vertex still in the queue — every such route would have to pay at least as much to get out there and then pay more, or at least nothing, to come back. If some $w < 0$, the assumption collapses: a long detour can end with a large rebate.

Figure 13.5 is the smallest counterexample. Three vertices, three edges, and — importantly — **no cycle at all**, so there is certainly no negative cycle and δ is perfectly well defined: $\delta(s, a) = 1$ by the route $\langle s, b, a \rangle$ costing $3 + (-2)$, and $\delta(s, b) = 3$.

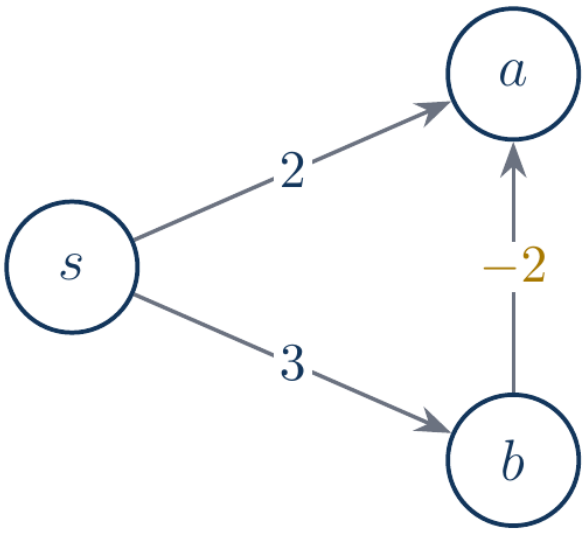


Figure 13.5 A graph on which Dijkstra silently returns the wrong answer. There is no cycle, so no negative cycle, and the true distances are $\delta(s, a) = 1$ and $\delta(s, b) = 3$. Dijkstra finalizes a at 2 — before it has ever looked at the edge (b, a) — and never revisits it.

Trace it. Initially $d[s] = 0$, $d[a] = d[b] = \infty$.

1. `ExtractMin` returns s with $d = 0$. Relax (s, a): $d[a] \leftarrow 2$. Relax (s, b): $d[b] \leftarrow 3$.
2. `ExtractMin` returns a with $d = 2$ (since $2 < 3$). a **joins** S **and is now final at** 2. It has no outgoing edges.

3. `ExtractMin` returns b with $d = 3$. Relax (b, a): the candidate value is $3 + (-2) = 1 < 2$. But a is already in S. A textbook Dijkstra either skips the update because a is finalized, or — worse — writes $d[a] = 1$ into an array that nobody will ever read again.

Either way the reported answer is $d[a] = 2$ and the truth is 1. The program of **Listing 13.1** runs exactly this and prints both algorithms' verdicts side by side:

Listing 13.1 Dijkstra and Bellman–Ford on a three-vertex graph with one negative edge and no cycle. Compile with `g++ -std=c++17 -O2 -o dijfail dijfail.cpp`.

```
#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>

using namespace std;

const long long INF = LLONG_MAX / 4;
struct Edge { int u, v; long long w; };

// Dijkstra, written so that a vertex is FINAL the moment it leaves the queue.
vector<long long> dijkstra(int n, const vector<Edge>& edges, int s, bool talk,
                           const vector<string>& name) {
    vector<vector<pair<int,long long>>> adj(n);
    for (const Edge& e : edges) adj[e.u].push_back({e.v, e.w});
    vector<long long> d(n, INF);
    vector<bool> done(n, false);
    d[s] = 0;
    priority_queue<pair<long long,int>, vector<pair<long long,int>>,
                   greater<pair<long long,int>>> Q;
    Q.push({0, s});
    while (!Q.empty()) {
        auto [key, u] = Q.top(); Q.pop();
        if (done[u]) continue;
        done[u] = true;                              // u joins S and is never revised
        if (talk) cout << "  finalize " << name[u] << " at d=" << d[u] << "\n";
        for (auto [v, w] : adj[u])
            if (!done[v] && d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
    }
    return d;
}

vector<long long> bellmanFord(int n, const vector<Edge>& edges, int s) {
    vector<long long> d(n, INF);
    d[s] = 0;
    for (int i = 1; i <= n - 1; ++i)
        for (const Edge& e : edges)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) d[e.v] = d[e.u] + e.w;
    return d;
}
```

Listing 13.1 (Continued) *Dijkstra and Bellman–Ford on a three-vertex graph with one negative edge and no cycle. Compile with g++ -std=c++17 -O2 -o dijfail dijfail.cpp.*

```
int main() {
    vector<string> name = {"s", "a", "b"};
    // s->a = 2, s->b = 3, b->a = -2. No cycle at all, so no negative cycle.
    vector<Edge> edges = {{0,1,2}, {0,2,3}, {2,1,-2}};

    cout << "Dijkstra on a graph with one negative edge:\n";
    vector<long long> dd = dijkstra(3, edges, 0, true, name);
    vector<long long> db = bellmanFord(3, edges, 0);

    cout << "\nvertex | Dijkstra | Bellman-Ford | truth\n";
    cout <<   "-------+----------+--------------+-------\n";
    const char* truth[] = {"0", "1", "3"};
    for (int v = 0; v < 3; ++v)
        cout << "   " << name[v] << "   |    " << dd[v]
             << "     |      " << db[v] << "       |   " << truth[v]
             << (dd[v] == db[v] ? "" : "   <-- Dijkstra is WRONG") << "\n";
    return 0;
}
```

```
Dijkstra on a graph with one negative edge:
  finalize s at d=0
  finalize a at d=2
  finalize b at d=3

vertex | Dijkstra | Bellman-Ford | truth
-------+----------+--------------+-------
   s   |    0     |      0       |   0
   a   |    2     |      1       |   1   <-- Dijkstra is WRONG
   b   |    3     |      3       |   3
```

> **WARNING.** Dijkstra does not detect this. It does not crash, it does not warn, it does not return an error code. It returns numbers that look entirely reasonable and are wrong. If you are not certain every weight is non-negative, do not use Dijkstra — use Bellman–Ford, which is slower and correct, or reweight the graph with Johnson's method of Chapter 15.

A popular "fix" is to add a large constant c to every edge weight so that all become non-negative. It does not work, and it is worth seeing why: a path of k edges gains kc, so paths with many edges are penalized more than paths with few, and the algorithm starts preferring short-hop routes that are not cheapest. Exercise 13.6 asks you to construct a graph where this fails.

13.7.4 The example, one step at a time

We now trace Dijkstra on a five-vertex graph, drawn in six panels. We follow every panel, and at each step we also write out the priority queue, which the figures imply but do not draw.

The graph has vertices s, u, v, x, y and ten directed edges:

$$w(s,u) = 10,\ w(s,x) = 5,\ w(u,v) = 1,\ w(u,x) = 2,\ w(x,u) = 3,$$

$$w(x,v) = 9,\ w(x,y) = 2,\ w(v,y) = 4,\ w(y,v) = 6,\ w(y,s) = 7.$$

In the drawing convention of these panels, a **white** vertex is still in Q, an **amber** vertex is the one just extracted, and a **navy** vertex is in S (solved). Bold arrows are the current predecessor edges, that is, the edges of G_π.

Panel (a) — after initialization. Every d is ∞ except $d[s] = 0$, and Q holds all five vertices.

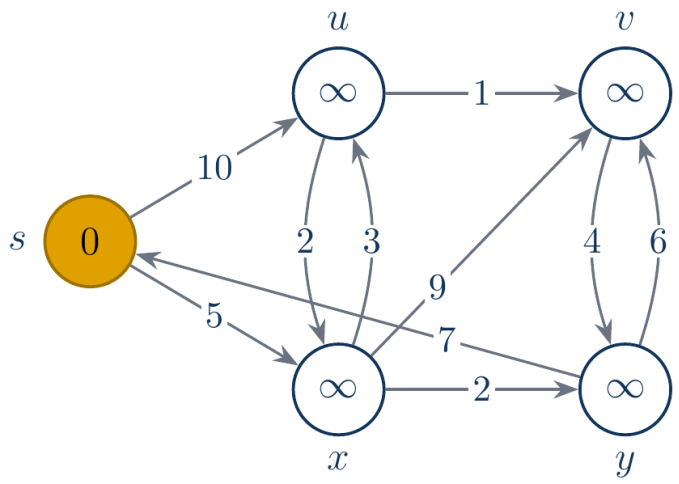


Figure 13.6 The Dijkstra example, panel (a): the graph immediately after `INITIALIZE-SINGLE-SOURCE`. $d[s] = 0$ (amber, about to be extracted) and every other estimate is ∞. $Q = \{s = 0,\ u = \infty,\ v = \infty,\ x = \infty,\ y = \infty\}$.

Step 1. `ExtractMin` returns s, the only vertex with a finite key. $S = \{s\}$. Relax the two edges out of s:

- (s, u), $w = 10$: $d[u] = \infty > 0 + 10 = 10$, so $d[u] \leftarrow 10$, $\pi[u] \leftarrow s$.
- (s, x), $w = 5$: $d[x] = \infty > 0 + 5 = 5$, so $d[x] \leftarrow 5$, $\pi[x] \leftarrow s$.

Now $Q = \{x = 5,\ u = 10,\ v = \infty,\ y = \infty\}$, which is the state drawn in **Figure 13.7**.

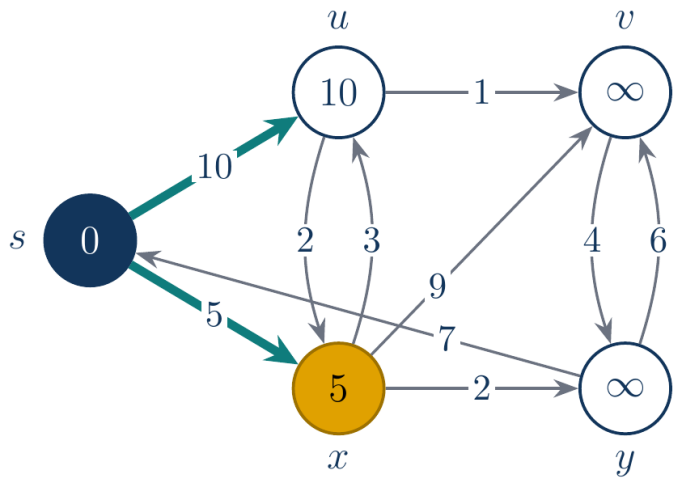


Figure 13.7 Panel (b): s is navy (solved), the two edges out of it have been relaxed, and x — amber — is the next vertex to be extracted because $5 < 10$. Note that u already has an estimate of 10; it will be improved before u is ever extracted.

Step 2. `ExtractMin` returns x with $d[x] = 5$. $S = \{s, x\}$. Relax the three edges out of x:

- (x, u), $w = 3$: $d[u] = 10 > 5 + 3 = 8$, so $d[u] \leftarrow 8$, $\pi[u] \leftarrow x$. **This is the `Decrease-Key` that the implementation note of §13.7.2 warns about**: u was sitting in Q with key 10 and must now sit there with key 8.
- (x, v), $w = 9$: $d[v] = \infty > 5 + 9 = 14$, so $d[v] \leftarrow 14$, $\pi[v] \leftarrow x$.
- (x, y), $w = 2$: $d[y] = \infty > 5 + 2 = 7$, so $d[y] \leftarrow 7$, $\pi[y] \leftarrow x$.

Now $Q = \{y = 7,\ u = 8,\ v = 14\}$ — see **Figure 13.8**.

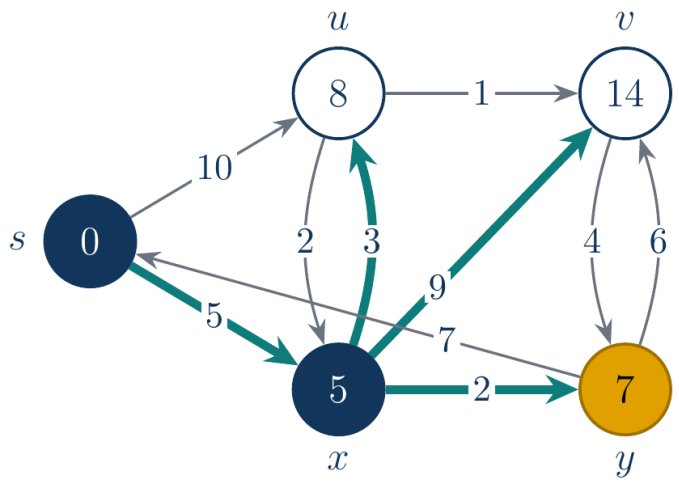


Figure 13.8 Panel (c): s and x are navy. The estimate for u has dropped from 10 to 8 and its predecessor edge has swung from (s, u) to (x, u) — the bold arrow now comes up from x. y is amber: at 7 it is the smallest key left.

Step 3. `ExtractMin` returns y with $d[y] = 7$. $S = \{s, x, y\}$. Relax the two edges out of y:

- (y, v), $w = 6$: $d[v] = 14 > 7 + 6 = 13$, so $d[v] \leftarrow 13$, $\pi[v] \leftarrow y$. Another `Decrease-Key`.
- (y, s), $w = 7$: $d[s] = 0$, and $7 + 7 = 14 > 0$, so nothing happens. A relaxation that changes nothing.

Now $Q = \{u = 8,\ v = 13\}$, as **Figure 13.9** shows.

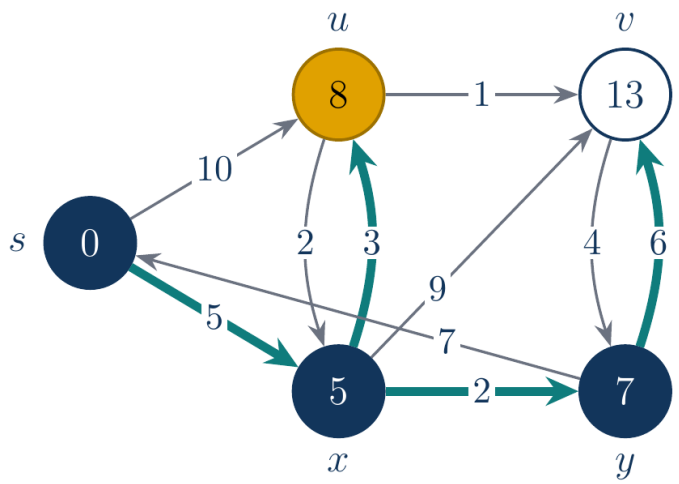


Figure 13.9 Panel (d): s, x and y are navy. v's estimate has improved from 14 to 13 and its bold predecessor arrow has swung from x to y. u, amber at 8, is next.

Step 4. `ExtractMin` returns u with $d[u] = 8$. $S = \{s, x, y, u\}$. Relax the two edges out of u:

- (u, v), $w = 1$: $d[v] = 13 > 8 + 1 = 9$, so $d[v] \leftarrow 9$, $\pi[v] \leftarrow u$. A third `Decrease-Key`; v's estimate has now been lowered three times, from ∞ to 14 to 13 to 9.
- (u, x), $w = 2$: $d[x] = 5$, and $8 + 2 = 10 > 5$, so nothing happens.

Now $Q = \{v = 9\}$; the picture is **Figure 13.10**.

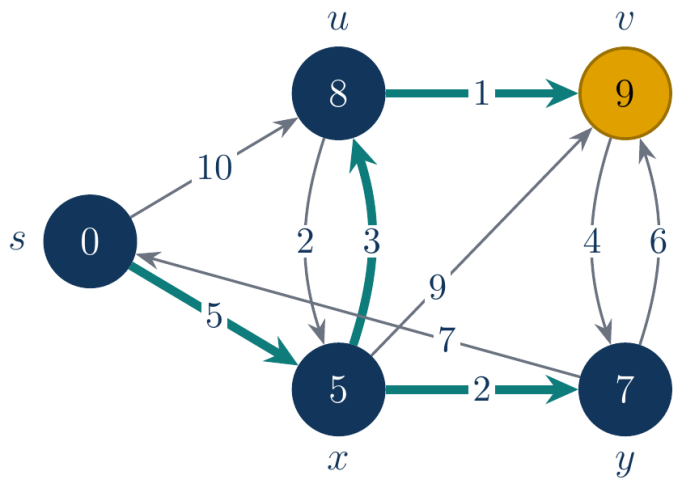


Figure 13.10 Panel (e): four vertices are navy, v is amber at 9, and its predecessor arrow now comes from u. Every edge in the graph has been relaxed at least once except the two leaving v.

Step 5. `ExtractMin` returns v with $d[v] = 9$. S is now all five vertices. Relax the one edge out of v:

- (v, y), $w = 4$: $d[y] = 7$, and $9 + 4 = 13 > 7$, so nothing happens.

Q is empty; the algorithm halts. **Figure 13.11** is the finished picture.

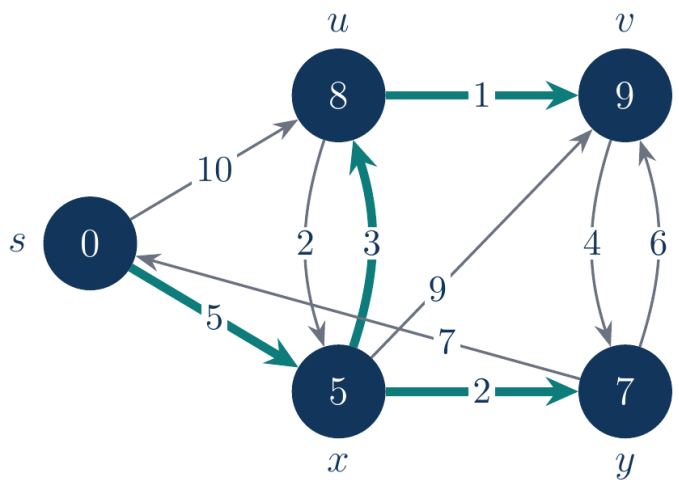


Figure 13.11 Panel (f): every vertex navy, the algorithm finished. The four bold edges (s, x), (x, u), (u, v) and (x, y) are the predecessor subgraph G_π — the shortest-path tree rooted at s. Reading a route off it is a matter of walking the bold arrows backward from any vertex to s.

13.7.5 The same trace as a table

Table 13.6 The complete run of Dijkstra on the example graph, one row per iteration of the `while` loop. The keys in Q only ever fall, while the extracted keys 0,5,7,8,9 never fall; §13.7.8 proves the second fact.

Step	Extracted u	$d[u]$	Edges relaxed (result)	Q After the step	S After
0	—	—	(initialize)	$s = 0, u = \infty, v = \infty, x = \infty, y = \infty$	$\emptyset$
1	s	0	$(s, u) \to 10$; $(s, x) \to 5$	$x = 5, u = 10, v = \infty, y = \infty$	$\{s\}$
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

The final answer is collected in **Table 13.7**.

Table 13.7 The output of Dijkstra on the example graph: distance, predecessor, and the route found by walking π backward. The d row agrees with **Table 13.1**, which found $\delta(s, v) = 9$ by listing six paths.

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

13.7.6 The program

Listing 13.2 is a complete C++ program. It builds the example graph, runs Dijkstra with a binary heap (the standard library's `priority_queue`), prints the trace in the same shape as **Table 13.6**, and finishes with the distances, the predecessors and the reconstructed routes.

One implementation decision deserves explanation before you read it. The C++ standard library's `priority_queue` has no `DecreaseKey`. Rather than write our own heap, we use the **lazy deletion** trick, exactly as Chapter 11 did for Prim: when $d[v]$ falls, we simply push a *second* copy of v into the queue with the new, smaller key, and leave the stale copy where it is. Copies are extracted smallest-first, so the good copy always comes out before the stale ones, and a stale copy is recognized on extraction because its key exceeds the current $d[v]$ (or because the vertex is already in S) and is discarded. The queue may hold up to $|E|$ entries instead of $|V|$, so the cost becomes $O(E\lg E) = O(E\lg V)$ — asymptotically the same as a proper binary-heap Dijkstra, since $\lg E < \lg V^2 = 2\lg V$.

Listing 13.2 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`.

```
#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 13.2 (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 13.2 (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;
}
```

13.7.7 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 that, line for line, with the six panels of §13.7.4 and with **Table 13.6**. The program is doing exactly what we did by hand, including the three `Decrease-Key` moments and the three relaxations that change nothing.

13.7.8 Correctness

This proof deserves to be taken slowly, because it is the deepest argument in the chapter and it is where the non-negativity assumption finally gets spent.

What is to be proved. *Whenever u is added to S, $d[u] = \delta(s,u)$, and equality is maintained thereafter.* In other words, the moment a vertex leaves the priority queue its estimate is already the true distance, and nothing that happens later disturbs it.

That is a *loop invariant*, in the sense of Chapter 2. Let us state it that way.

THEOREM 13.10 (Correctness of Dijkstra). Run `Dijkstra(G, w, s)` on a weighted digraph with $w(u,v) \geq 0$ for every edge. At the top of each iteration of the `while` loop of line 6, $d[v] = \delta(s,v)$ for every $v \in S$. Consequently, at termination $d[v] = \delta(s,v)$ for every $v \in V$, and G_π is a shortest-path tree rooted at s.

Proof. The proof is by contradiction, and it runs in the picture of **Figure 13.12**, which we shall refer to three times.

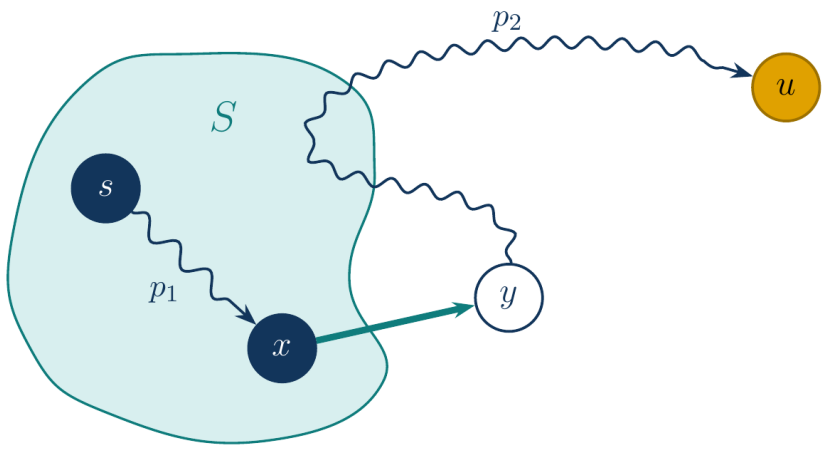


Figure 13.12 The picture behind Dijkstra's correctness proof: the light-teal region is the solved set S, and u is the first vertex allegedly extracted with a wrong estimate. The true shortest path leaves S at the edge (x,y), so it reads $s \overset{p_1}{\rightsquigarrow} x \to y \overset{p_2}{\rightsquigarrow} u$.

Setting up the contradiction. By Lemma 13.5 we already know $d[v] \geq \delta(s,v)$ for every v at every moment — in symbols, $\forall v,\ d[v] \geq \delta(s,v)$. Suppose the theorem is false. Then at some extraction the invariant breaks. Let u be the **first** vertex ever extracted for which $d[u] \neq \delta(s,u)$; by the upper-bound property this means

$$d[u] > \delta(s,u). \tag{13.8}$$

The word *first* is doing a great deal of work: it means every vertex already in S has its correct value, which is the fact we shall use twice.

Step 1: $u \neq s$. When s is extracted (necessarily first, since $d[s] = 0$ and everything else is $\infty \geq 0$), $d[s] = 0 = \delta(s,s)$, so the invariant holds for s and u cannot be s.

Step 2: u is reachable from s. If it were not, $\delta(s,u) = \infty$, and by Lemma 13.6 $d[u] = \infty$ too, contradicting Equation (13.8), which requires a strict inequality. So a shortest path p from s to u exists.

Step 3: decompose p at the boundary of S. Consider the moment just before u is extracted. At that moment $s \in S$ and $u \notin S$. Walk along p from s toward u and let y be the **first** vertex on p that is not in S, and let x be its predecessor on p, so $x \in S$. (Possibly $x = s$; possibly $y = u$. Neither special case harms the argument.) This is exactly what **Figure 13.12** draws: p splits as

$$s \overset{p_1}{\rightsquigarrow} x \ \to\ y \overset{p_2}{\rightsquigarrow} u.$$

Step 4: $d[y] = \delta(s, y)$. This is the heart of the proof, and it takes two observations.

First, $x \in S$ and x was extracted *before* u; since u is the first vertex extracted with a wrong estimate, x's estimate is right: $d[x] = \delta(s, x)$.

Second, when the algorithm inserted x into S at line 8, it immediately relaxed every edge out of x at lines 9–11 — including the edge (x, y). At that moment $d[x]$ already equalled $\delta(s, x)$. And p is a shortest path through the edge (x, y), so by optimal substructure (Theorem 13.1) its prefix $s \rightsquigarrow x \to y$ is a shortest path to y, i.e. $\delta(s, y) = \delta(s, x) + w(x, y)$. Those are precisely the hypotheses of the convergence property, Lemma 13.7. Applying it: after that relaxation, $d[y] = \delta(s, y)$, and by Lemma 13.5 it has not moved since.

Step 5: $\delta(s, y) \leq \delta(s, u)$ *— and this is where non-negativity is spent.* The subpath p_2 runs from y to u and, by optimal substructure again, is a shortest path, so $w(p_2) = \delta(y, u)$. Decomposing p,

$$\delta(s, u) \;=\; \delta(s, y) + w(p_2) \;=\; \delta(s, y) + \delta(y, u).$$

Every edge of p_2 has non-negative weight, so $w(p_2) \geq 0$, and therefore

$$\delta(s, y) \;\leq\; \delta(s, u). \tag{13.9}$$

> **This is the only step that uses** $w \geq 0$, and everything depends on it. If p_2 could have negative total weight then $\delta(s, y)$ could *exceed* $\delta(s, u)$: the true route to u would pass through a vertex that is *further away* than u itself, and a greedy algorithm that always takes the nearest vertex would never find it. That is exactly what happens in **Figure 13.5**, where the true route to a (distance 1) passes through b (distance 3).

Step 6: the contradiction. Chain the inequalities. Using Step 4 for the equality, then Equation (13.9), then Equation (13.8):

$$d[y] \;=\; \delta(s, y) \;\leq\; \delta(s, u) \;<\; d[u].$$

So $d[y] < d[u]$ strictly. But both y and u were in the priority queue at the moment of extraction, and the algorithm extracted u — the vertex of *minimum* key. That requires $d[u] \leq d[y]$. The two statements contradict each other. In one line: *but* $d[u] > d[y] \Rightarrow$ *the algorithm would have chosen* y *from the priority queue to process next, not* u *— contradiction.*

Therefore no such u exists, and every vertex is extracted with $d[u] = \delta(s, u)$. Since every vertex is extracted exactly once and the loop runs until Q is empty, at termination $d[v] = \delta(s, v)$ for all v, and Lemma 13.9 then says G_π is a shortest-path tree. ■

A corollary that is useful in practice.

> **COROLLARY 13.11.** The sequence of keys returned by successive `Extract-Min` calls in Dijkstra is non-decreasing.

Proof. It is enough to compare two *consecutive* extractions, since a sequence in which each term is at least the one before it is non-decreasing. So let u be extracted in one iteration and u' in the next, and follow the keys through the iteration in between, one step at a time.

1. When u is extracted, it has the smallest key in Q. So every vertex v still in Q has $d[v] \geq d[u]$.
2. The iteration then relaxes the edges out of u. A relaxation of (u, v) either changes nothing, or sets $d[v] \leftarrow d[u] + w(u, v)$. Because $w(u, v) \geq 0$, the new value is still at least $d[u]$.

3. So at the end of the iteration every key in Q is still at least $d[u]$ — the untouched keys by step 1, the lowered ones by step 2.
4. The next extraction takes one of those keys, so $d[u'] \geq d[u]$. ■

Notice that step 2 is the only place the argument uses $w \geq 0$. With a negative edge, a relaxation could plant a key *below* $d[u]$, and the next extraction would go backward — which is exactly what happens in the Profitina Laundry cost graph of §13.12.2.

The practical use: in the trace of §13.7.4 the extracted keys were 0,5,7,8,9 — monotone, as promised. This also justifies the **early exit** for the single-pair problem: if you only want $\delta(s,t)$, you may stop the loop the moment t is extracted, because every key extracted afterward is at least as large and no later relaxation can lower $d[t]$.

13.7.9 Running time

The accounting is refreshingly simple, because the algorithm's whole cost lives in the priority queue.

Count the operations. Line 5 inserts every vertex, so there are $|V|$ `Insert` operations. The `while` loop runs once per vertex, so there are $|V|$ `Extract-Min` operations. Lines 9–11 examine each edge (u,v) exactly once, when its tail u is extracted — across the whole run, $\sum_{u \in V} |Adj[u]| = |E|$ examinations — and each may cause one `Decrease-Key`. So there are at most $|E|$ `Decrease-Key` operations. Everything else is $O(V+E)$ bookkeeping. The total is therefore

$$T = |V| \cdot T_{\text{Extract-Min}} \;+\; |E| \cdot T_{\text{Decrease-Key}}. \tag{13.10}$$

Now substitute the three queue implementations from Chapter 11 into Equation (13.10), as **Table 13.8** does.

Table 13.8 Dijkstra's running time for three priority-queue implementations. The binary-heap total simplifies to $O(E\lg V)$ when every vertex is reachable from s, so that $E \geq V-1$; the Fibonacci-heap row is due to Fredman and Tarjan [5].

Q Implemented as	T(Extract-Min)	T(Decrease-Key)	Total
unsorted array	$\Theta(V)$	$\Theta(1)$	$\Theta(V^2)$
binary heap	$\Theta(\lg V)$	$\Theta(\lg V)$	$O((V+E)\lg V) = O(E\lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$\Theta(1)$ amortized	$O(V\lg V + E)$

Read the table as advice, not trivia.

Unsorted array, $\Theta(V^2)$. `Extract-Min` is a linear scan; `Decrease-Key` is one array write. Total $|V| \cdot \Theta(V) + |E| \cdot \Theta(1) = \Theta(V^2 + E) = \Theta(V^2)$, since $E \leq V^2$. This is the *right* choice for a **dense** graph, where $E = \Theta(V^2)$ anyway: you cannot beat $\Theta(V^2)$ when you must look at $\Theta(V^2)$ edges, and the array has no heap overhead at all.

Binary heap, $O(E\lg V)$. The standard choice for a **sparse** graph. With $E = \Theta(V)$ this is $O(V\lg V)$, hugely better than $\Theta(V^2)$. This is what **Listing 13.2** implements, via the standard library's `priority_queue` and lazy deletion.

Fibonacci heap, $O(V\lg V + E)$. The best bound, because `Decrease-Key` — the operation performed up to $|E|$ times — becomes constant. It is a theoretical triumph and a practical disappointment: the constants hidden in a Fibonacci heap are large enough that for graphs of realistic size a binary heap is usually faster in wall-clock terms. Use it when E is very large relative to V and you have measured, not before.

The crossover between array and heap is at roughly $E = \Theta(V^2/\lg V)$. Below that the heap wins; above it the array does.

13.7.10 Measured

Asymptotics are predictions. **Listing 13.3** tests them, and at the same time tests something more important: that this chapter's algorithms agree with each other. It runs three implementations — heap Dijkstra, array Dijkstra, Bellman–Ford — on the example graph and on two thousand randomly generated non-negative digraphs, checks that all three produce identical distance vectors every time, and then times the two Dijkstras on sparse and dense graphs of increasing size.

Listing 13.3 A cross-check and a benchmark: heap Dijkstra, array Dijkstra and Bellman–Ford must agree on every non-negative graph, and the two Dijkstras are then timed on sparse ($m = 8n$) and dense ($m = n^2/4$) inputs.

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <queue>
#include <random>
#include <chrono>
#include <climits>

using namespace std;
using namespace std::chrono;

const long long INF = LLONG_MAX / 4;
struct Edge { int u, v; long long w; };

vector<long long> dijkstraHeap(int n, const vector<Edge>& edges, int s) {
    vector<vector<pair<int,long long>>> adj(n);
    for (const Edge& e : edges) adj[e.u].push_back({e.v, e.w});
    vector<long long> d(n, INF);
    d[s] = 0;
    priority_queue<pair<long long,int>, vector<pair<long long,int>>,
                   greater<pair<long long,int>>> Q;
    Q.push({0, s});
    while (!Q.empty()) {
        auto [key, u] = Q.top(); Q.pop();
        if (key > d[u]) continue;                  // stale 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;
}

// Dijkstra with a plain array instead of a heap: EXTRACT-MIN is a linear scan.
vector<long long> dijkstraArray(int n, const vector<Edge>& edges, int s) {
    vector<vector<pair<int,long long>>> adj(n);
    for (const Edge& e : edges) adj[e.u].push_back({e.v, e.w});
    vector<long long> d(n, INF);
    vector<bool> done(n, false);
    d[s] = 0;
    for (int it = 0; it < n; ++it) {
        int u = -1;
        for (int v = 0; v < n; ++v)                // the O(V) EXTRACT-MIN
            if (!done[v] && (u == -1 || d[v] < d[u])) u = v;
        if (u == -1 || d[u] >= INF) break;
        done[u] = true;
        for (auto [v, w] : adj[u])
            if (d[u] + w < d[v]) d[v] = d[u] + w;  // DECREASE-KEY is O(1)
```

Listing 13.3 (Continued) *A cross-check and a benchmark: heap Dijkstra, array Dijkstra and Bellman–Ford must agree on every non-negative graph, and the two Dijkstras are then timed on sparse (m = 8n) and dense (m = n^2/4) inputs.*

```
    }
    return d;
}

vector<long long> bellmanFord(int n, const vector<Edge>& edges, int s) {
    vector<long long> d(n, INF);
    d[s] = 0;
    for (int i = 1; i <= n - 1; ++i) {
        bool changed = false;
        for (const Edge& e : edges)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) {
                d[e.v] = d[e.u] + e.w; changed = true;
            }
        if (!changed) break;
    }
    return d;
}

int main() {
    // (1) the Dijkstra example graph: all three agree
    vector<Edge> sample = {{0,1,10},{0,3,5},{1,2,1},{1,3,2},
                           {3,1,3},{3,2,9},{3,4,2},{2,4,4},{4,2,6},{4,0,7}};
    auto a1 = dijkstraHeap(5, sample, 0);
    auto a2 = dijkstraArray(5, sample, 0);
    auto a3 = bellmanFord(5, sample, 0);
    cout << "example graph  heap=";
    for (long long x : a1) cout << x << " ";
    cout << " array=";
    for (long long x : a2) cout << x << " ";
    cout << " BF=";
    for (long long x : a3) cout << x << " ";
    cout << ((a1 == a2 && a2 == a3) ? " -> all agree\n\n" : " -> DISAGREE\n\n");

    // (2) 2000 random non-negative digraphs: the three must always agree
    mt19937 rng(12345);
    int bad = 0;
    for (int trial = 0; trial < 2000; ++trial) {
        int n = 2 + (int)(rng() % 9);
        int m = (int)(rng() % (n * n));
        vector<Edge> e;
        for (int i = 0; i < m; ++i)
            e.push_back({(int)(rng() % n), (int)(rng() % n), (long long)(rng() % 20)});
        auto h = dijkstraHeap(n, e, 0), r = dijkstraArray(n, e, 0), b = bellmanFord(n, e,
0);
        if (!(h == r && r == b)) ++bad;
    }
    cout << "2000 random non-negative digraphs: " << bad << " disagreements\n\n";
```

Listing 13.3 (Continued) *A cross-check and a benchmark: heap Dijkstra, array Dijkstra and Bellman–Ford must agree on every non-negative graph, and the two Dijkstras are then timed on sparse (m = 8n) and dense (m = n^2/4) inputs.*

```
    // (3) how the two Dijkstras scale, on a sparse and on a dense graph
    cout << "        n       m    array(ms)   heap(ms)\n";
    for (int n : {1000, 2000, 4000}) {
        for (int deg : {8, 0}) {                    // deg 0 means "dense": m = n*8 or n*n/4
            long long m = deg ? (long long)n * deg : (long long)n * n / 4;
            vector<Edge> e;
            e.reserve(m);
            for (long long i = 0; i < m; ++i)
                e.push_back({(int)(rng() % n), (int)(rng() % n), (long long)(rng() %
1000)});
            auto t0 = steady_clock::now();
            auto r = dijkstraArray(n, e, 0);
            auto t1 = steady_clock::now();
            auto h = dijkstraHeap(n, e, 0);
            auto t2 = steady_clock::now();
            if (r != h) { cout << "  DISAGREE\n"; return 1; }
            cout << setw(9) << n << setw(8) << m
                 << setw(11) << fixed << setprecision(1)
                 << duration<double, milli>(t1 - t0).count()
                 << setw(11) << duration<double, milli>(t2 - t1).count() << "\n";
        }
    }
    return 0;
}
```

```
example graph  heap=0 8 9 5 7  array=0 8 9 5 7  BF=0 8 9 5 7  -> all agree

2000 random non-negative digraphs: 0 disagreements

        n       m    array(ms)   heap(ms)
     1000    8000        2.1        0.5
     1000  250000        7.2        4.5
     2000   16000        6.9        1.0
     2000 1000000       27.9       18.4
     4000   32000       25.9        2.1
     4000 4000000      113.7       83.8
```

Read the table. On the **sparse** rows ($m = 8n$), the array version grows like n^2: from $n = 1000$ to $n = 4000$ is a factor of four in n and 2.1 → 25.9 ms, a factor of about 12 — close to the predicted 16, the shortfall being cache effects at small sizes. The heap version over the same range goes 0.5 → 2.1 ms, a factor of about 4, matching the predicted $O(E\lg V) = O(n\lg n)$ almost exactly. On the sparse graphs the heap is between four and twelve times faster, and the gap widens with n. (Timings are from one run on one machine; yours will differ in the absolute numbers, but the ratios are what the analysis predicts.)

On the **dense** rows ($m = n^2/4$) the two are much closer — 7.2 against 4.5, then 27.9 against 18.4, then 113.7 against 83.8 — because both are now dominated by the $\Theta(E)$ edge scan rather than by the queue. The predicted crossover has not quite been reached at these sizes, but the trend is exactly the one the table predicts: the heap's advantage shrinks as the graph fills in.

And the first two lines are the ones that matter most for trusting anything else in this chapter: on the example graph and on two thousand random graphs, three independent implementations produced the same numbers every time.

13.7.11 Dijkstra, Prim, and breadth-first search

Three algorithms in this book have the same skeleton. Putting them side by side, as **Table 13.9** does, is the fastest way to stop confusing them.

Table 13.9 Breadth-first search, Prim and Dijkstra compared. The single line 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 together, so it never needs the triangle inequality.

	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 last row of that table repays thought. Prim is safe with negative weights because it never forms a sum $d[u] + w(u,v)$; it only ever compares individual edge weights, and comparison is indifferent to sign. Dijkstra's greed rests on the claim that extending a path cannot shorten it, which is a statement about sums. Same skeleton, different bones.

13.8 The Bellman–Ford Algorithm

13.8.1 Why we need it

The starting point is a plain statement: *Dijkstra's algorithm doesn't work when there are negative edges.* And then, in one sentence, the reason: *we cannot be greedy on the assumption that the lengths of paths will only increase in the future.* Dijkstra's whole method is to finalize the nearest unsolved vertex and never look back, and §13.7.3 showed a three-vertex graph where that is wrong.

The Bellman–Ford algorithm — due independently to Richard Bellman [6] in 1958 and Lester Ford [7] in 1956, with Edward Moore [8] arriving at the same method in 1957 — abandons greed entirely. It does not try to be clever about the order in which it relaxes edges. It relaxes **every edge,** $|V| - 1$ **times**, and relies on the path-relaxation property (Lemma 13.8) to guarantee that the right order happened somewhere inside all that repetition. In exchange for being slower it gets two things Dijkstra cannot offer: it is correct with negative edges, and it can **detect** a negative cycle and say so, rather than returning nonsense.

13.8.2 The algorithm

Figure 13.13 gives the 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 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.

Line by line:

Lines 01–04. `INITIALIZE-SINGLE-SOURCE`, exactly **Pseudocode 13.1**: every estimate ∞, every predecessor NIL, and the source 0. **Line 05.** Repeat the body $|V| - 1$ times. Not $|V|$; not $|E|$. The count comes from Theorem 13.4: no shortest path has more than $|V| - 1$ edges. **Line 06.** For **every** edge of the graph, in whatever order the edge list happens to give. **Line 07.** `Relax(u,v,w)` — the same three lines as everywhere else in this chapter. **Lines 08–09.** The detector. Sweep the edges one final time. If any edge (u, v) still satisfies $d[v] > d[u] + w(u, v)$, return `false`. **Line 10.** Otherwise return `true`: no reachable negative cycle, and d holds the true distances.

> **KEY IDEA.** Bellman–Ford makes no decision at all. It does not choose which edge to relax next, so it needs no priority queue, no ordering, and no assumption about signs. Its only idea is repetition, and the counting argument of §13.8.3 says how much repetition is enough.

13.8.3 Why $|V| - 1$ passes are enough

> **THEOREM 13.12.** Let G have no negative-weight cycle reachable from s. After the i-th pass of lines 05–07, $d[v] = \delta(s, v)$ for every vertex v whose shortest path from s uses at most i edges. Consequently, after $|V| - 1$ passes, $d[v] = \delta(s, v)$ for every v.

Proof. Fix a vertex v that is reachable from s. By Theorem 13.4, there is a *simple* shortest path from s to v; call it

$$p = \langle v_0, v_1, \ldots, v_k \rangle, \qquad v_0 = s,\ v_k = v,$$

and since p is simple it has $k \le |V| - 1$ edges.

Now apply the path-relaxation property, Lemma 13.8. It requires only that the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ be relaxed **in that relative order**, with anything at all in between. Look at what Bellman–Ford does: pass 1 relaxes every edge of G, so in particular it relaxes (v_0, v_1) at some point. Pass 2 relaxes every edge again, so it relaxes (v_1, v_2). Pass i relaxes (v_{i-1}, v_i). After k passes, all k edges of p have been relaxed in the required order, so Lemma 13.8 gives $d[v_k] = \delta(s, v_k)$ — and by Lemma 13.5 it stays so.

Since $k \leq |V| - 1$ for every v, running $|V| - 1$ passes suffices for all of them simultaneously. ■

> **Read that again.** The proof does *not* claim that pass i relaxes the i-th edge of p and nothing else useful. It claims only that pass i relaxes *at least* the i-th edge, somewhere among the $|E|$ relaxations it performs. Often a single pass makes far more progress — if the edge list happens to be in the right order, one pass can solve the whole graph. The bound $|V| - 1$ is the worst case, achieved when the edge list is in exactly the wrong order.

A second proof runs the induction a slightly different way, through the cheapest paths with a bounded number of edges, and it is worth having both.

> **THEOREM 13.12′ (edge-count form).** Let $\delta_i(s, u)$ denote the weight of the cheapest path from s to u *among all paths using at most i edges* (∞ if there is none). If no negative-weight cycle is reachable from s, then after the i-th pass of Bellman–Ford, $\delta(s, u) \leq d[u] \leq \delta_i(s, u)$ for every u.

Proof, by induction on i.

Base case, $i = 0$. Before any pass, $d[s] = 0$ and $d[u] = \infty$ for $u \neq s$. And $\delta_0(s, s) = 0$ (the empty path) while $\delta_0(s, u) = \infty$ for $u \neq s$ (no path of zero edges reaches anywhere else). They agree, so $d[u] \leq \delta_0(s, u)$ holds with equality. This base case really is trivial.

Inductive step. Assume $d[u] \leq \delta_{i-1}(s, u)$ for all u after pass $i - 1$. Fix a vertex u and consider a cheapest path to u using at most i edges. The key observation is that there are exactly two possibilities:

- **Either** it uses at most $i - 1$ edges, in which case $\delta_i(s, u) = \delta_{i-1}(s, u)$, and $d[u]$ is already at most that value and no relaxation ever increases it;
- **or** it uses exactly i edges, in which case strip off its last edge (z, u): what remains is a path to z of at most $i - 1$ edges, and by optimal substructure it is a cheapest such path, so $\delta_i(s, u) = \delta_{i-1}(s, z) + w(z, u)$.

The two cases together say that δ_i obeys the recurrence

$$\delta_i(s, u) \;=\; \min\Big(\delta_{i-1}(s, u),\; \min_{(z,u) \in E} (\delta_{i-1}(s, z) + w(z, u))\Big). \tag{13.11}$$

Now look at pass i. It relaxes every edge, so in particular it relaxes the edge (z, u) of the second case. At that moment $d[z] \leq \delta_{i-1}(s, z)$: that held after pass $i - 1$, and $d[z]$ can only have fallen since. After the relaxation, therefore, $d[u] \leq d[z] + w(z, u) \leq \delta_{i-1}(s, z) + w(z, u) = \delta_i(s, u)$. In the first case $d[u] \leq \delta_{i-1}(s, u) = \delta_i(s, u)$ already. Either way $d[u] \leq \delta_i(s, u)$ at the end of pass i, which is the right-hand inequality; the left-hand one, $d[u] \geq \delta(s, u)$, is the upper-bound property, Lemma 13.5. In words: *each pass tries to relax every edge — (z, u) included — so it catches both cases of* Equation (13.11). ■

Why an inequality and not an equality? Because a single pass may do *more* than one edge's worth of work: if the edges of a path happen to appear in the list in the right order, one pass can carry an estimate along several edges at once, and then $d[u]$ drops below $\delta_i(s, u)$ — it is already the true

distance. The trace of §13.8.4 shows this in pass 2, where $d[x]$ reaches 4 along the two-edge route $s \to y \to x$.

Finally the loop closes: after $n-1$ passes, $\delta(s,u) \le d[u] \le \delta_{n-1}(s,u)$ for each vertex u, and since any shortest path has at most $n-1$ edges (Theorem 13.4), $\delta_{n-1}(s,u) = \delta(s,u)$. The two ends of the inequality meet, so $d[u] = \delta(s,u)$ and we are done.

13.8.4 The example, pass by pass

We now trace Bellman–Ford on a five-vertex graph with three negative edges. The vertices are s, t, x, y, z and the ten edges are

$$w(t,x) = 5,\ w(t,y) = 8,\ w(t,z) = -4,\ w(x,t) = -2,\ w(y,x) = -3,$$

$$w(y,z) = 9,\ w(z,x) = 7,\ w(z,s) = 2,\ w(s,t) = 6,\ w(s,y) = 7.$$

The order in which the edges are relaxed matters for the *trace*, though not for the answer it ends with, and we use the order written above: (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y).

Notice that the two edges out of s come **last** in the list. That is deliberately the worst case: nothing can happen in a pass until s's own edges are relaxed, so each pass pushes the frontier forward by exactly one vertex, and all four passes are needed.

Panel (a) — after initialization. $d[s] = 0$, everything else ∞.

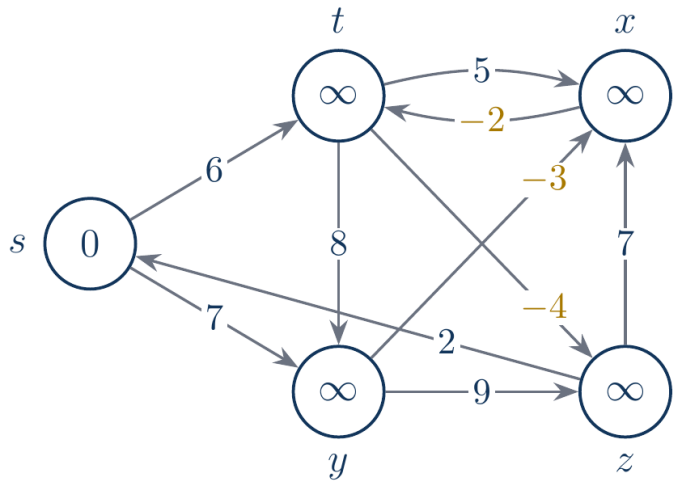


Figure 13.14 The Bellman–Ford example, panel (a): initialization. Three of the ten edges carry negative weights — $w(x,t) = -2$, $w(y,x) = -3$, $w(t,z) = -4$ — which is exactly why Dijkstra may not be used here. There is no negative *cycle*, however: check $t \to z \to s \to t$, which costs $-4+2+6 = 4 > 0$, and $t \to x \to t$, which costs $5 + (-2) = 3 > 0$.

Pass 1. Relax all ten edges in order. The first eight do nothing, because $d[t]$, $d[x]$, $d[y]$ and $d[z]$ are all still ∞. Then:

- $(s,t), w = 6$: $d[t] = \infty > 0 + 6$, so $d[t] \leftarrow 6$, $\pi[t] \leftarrow s$.
- $(s,y), w = 7$: $d[y] = \infty > 0 + 7$, so $d[y] \leftarrow 7$, $\pi[y] \leftarrow s$.

After pass 1: $d = (s{:}\,0,\ t{:}\,6,\ x{:}\,\infty,\ y{:}\,7,\ z{:}\,\infty)$ — the state of **Figure 13.15**.

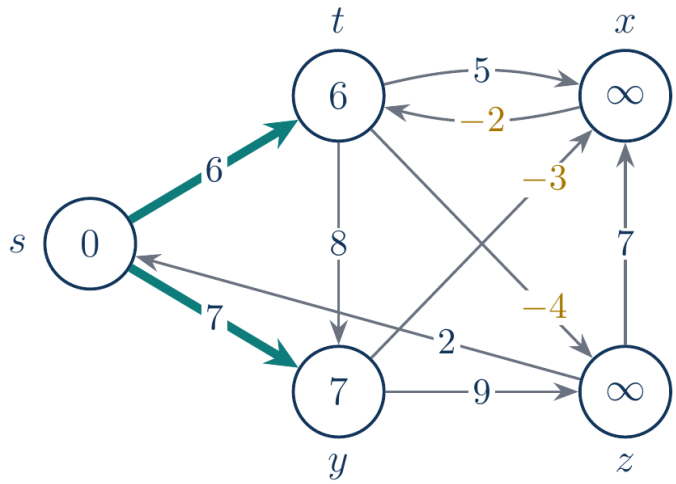


Figure 13.15 Panel (b): after pass 1. Only the two edges leaving s have had any effect, because they are the only ones whose tail had a finite estimate when they were examined. The bold arrows are the current predecessor edges.

Pass 2. Now $d[t]$ and $d[y]$ are finite, so more edges bite:

- $(t, x), w = 5$: $d[x] = \infty > 6 + 5 = 11$, so $d[x] \leftarrow 11, \pi[x] \leftarrow t$.
- $(t, y), w = 8$: $6 + 8 = 14 > 7$ — no change.
- $(t, z), w = -4$: $d[z] = \infty > 6 + (-4) = 2$, so $d[z] \leftarrow 2, \pi[z] \leftarrow t$.
- $(x, t), w = -2$: $11 + (-2) = 9 > 6$ — no change.
- $(y, x), w = -3$: $d[x] = 11 > 7 + (-3) = 4$, so $d[x] \leftarrow 4, \pi[x] \leftarrow y$. **The estimate for x has been improved twice in the same pass.**
- $(y, z), w = 9$: $7 + 9 = 16 > 2$ — no change.
- $(z, x), w = 7$: $2 + 7 = 9 > 4$ — no change.
- $(z, s), w = 2$: $2 + 2 = 4 > 0$ — no change.
- $(s, t), (s, y)$: no change.

After pass 2: $d = (0,\ 6,\ 4,\ 7,\ 2)$, drawn in **Figure 13.16**.

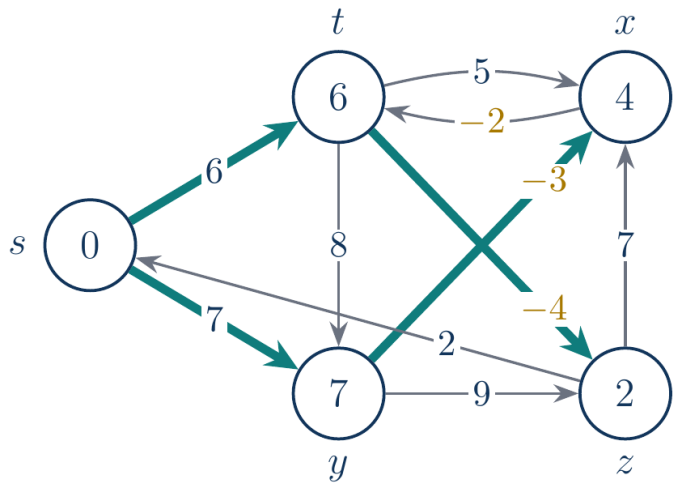


Figure 13.16 Panel (c): after pass 2. Every vertex now has a finite estimate. $d[x]$ settled at 4 via y and the negative edge (y, x), and $d[z]$ at 2 via t and the negative edge (t, z). Two of the five estimates are still wrong.

Pass 3. Only one edge does anything:

- $(x, t), w = -2$: $d[t] = 6 > 4 + (-2) = 2$, so $d[t] \leftarrow 2, \pi[t] \leftarrow x$.

Everything else is already at least as good. After pass 3: $d = (0,\ 2,\ 4,\ 7,\ 2)$, drawn in **Figure 13.17**.

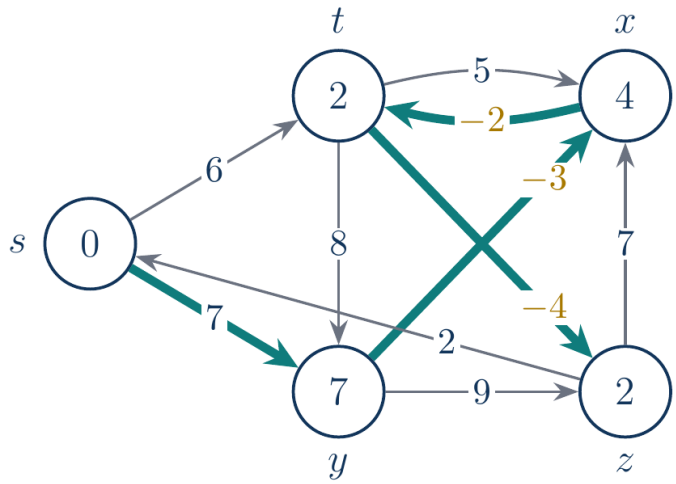


Figure 13.17 Panel (d): after pass 3. The estimate for t has fallen from 6 to 2, because the route $s \to y \to x \to t$ costs $7 + (-3) + (-2) = 2$, beating the direct edge of weight 6. Its bold predecessor arrow has swung from s to x. This improvement was invisible until pass 3 because it needs three edges.

Pass 4. Again exactly one edge does anything:

- $(t, z), w = -4$: $d[z] = 2 > 2 + (-4) = -2$, so $d[z] \leftarrow -2, \pi[z]$ stays t.

After pass 4: $d = (0,\ 2,\ 4,\ 7,\ -2)$, which is **Figure 13.18**; these are the true distances.

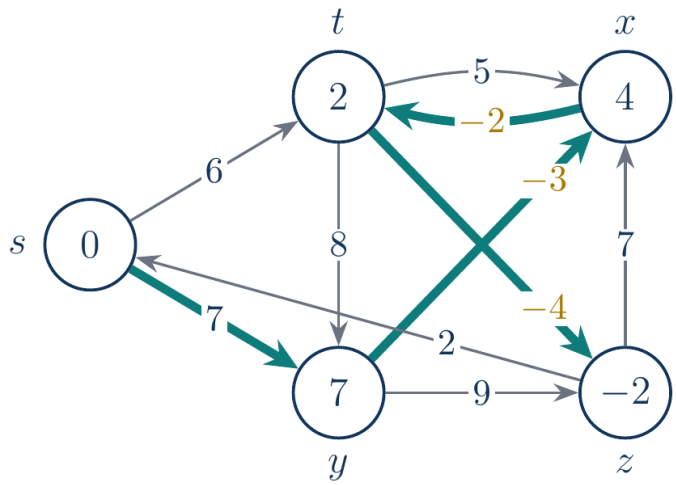


Figure 13.18 Panel (e), after pass 4: $\delta(s, z) = -2$ by the four-edge route $s \to y \to x \to t \to z$, of weight $7 + (-3) + (-2) + (-4)$. Four edges is the most a simple path in a five-vertex graph can have, so all $|V| - 1 = 4$ passes were needed.

The detector pass. Sweep all ten edges once more. Check the ones that could conceivably fire: (y, x) gives $7 - 3 = 4 = d[x]$, not less; (x, t) gives $4 - 2 = 2 = d[t]$, not less; (t, z) gives $2 - 4 = -2 = d[z]$, not less; (z, s) gives $-2 + 2 = 0 = d[s]$, not less. Nothing improves. The algorithm returns `true`.

13.8.5 The same trace as a table

Table 13.10 The complete run of Bellman–Ford on the example graph, one row per pass over all ten edges. In pass 2, $d[x]$ was set to 11 and then to 4 *within the same pass*; 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`

The final answer, with the routes read off π, is **Table 13.11**.

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), a structure a greedy method that finalizes vertices early can easily miss.

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

13.8.6 The program

Listing 13.4 is the whole algorithm, printing each pass so that its output can be checked line by line against the five panels above.

Listing 13.4 Bellman–Ford with negative-cycle detection on the example graph, printing every pass. The second half of `main` 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 shows the detector firing. Compile with `g++ -std=c++17 -O2 -o bellman bellman.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 13.4 (Continued) *Bellman–Ford with negative-cycle detection on the example graph, printing every pass. The second half of main 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 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 13.4 (Continued) *Bellman–Ford with negative-cycle detection on the example graph, printing every pass. The second half of main 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 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;
}
```

Two details in that program are worth pausing over.

The guard `d[e.u] < INF` before every relaxation is the no-path property (Lemma 13.6) turned into code. Without it, an edge out of an unreached vertex would compute $\mathrm{INF} + w$, which for a negative w is *less* than INF, and the algorithm would cheerfully propagate garbage distances across the unreachable part of the graph. Using `LLONG_MAX/4` rather than `LLONG_MAX` means that even if the guard were forgotten the arithmetic would not overflow — belt and braces.

The `if (!changed) break;` is a free optimization with a proof behind it: if a whole pass relaxes nothing, then no d value changed, so the next pass would examine exactly the same values and also change nothing, and so would every pass after that. Stopping early cannot alter the result. On well-ordered edge lists this turns the worst-case $|V| - 1$ passes into two or three. Note that the program still runs the *detector* pass afterward — skipping that would lose the negative-cycle test.

13.8.7 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 of §13.8.4, including the transient $d[x] = 11$ inside pass 2.

13.8.8 Negative-cycle detection, and why the extra pass finds it

Lines 08–09 of **Figure 13.13** are four lines of code carrying a theorem.

> **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.

Proof, both directions.

If there is no reachable negative cycle, it returns `true`. By Theorem 13.12, after $|V| - 1$ passes $d[v] = \delta(s, v)$ for every v. Take any edge (u, v). If $d[u] = \infty$ the guard skips it. Otherwise, by the edge form of the triangle inequality (Corollary 13.3),

$$d[v] = \delta(s, v) \;\leq\; \delta(s, u) + w(u, v) = d[u] + w(u, v),$$

so the test $d[v] > d[u] + w(u, v)$ on line 09 is false. No edge fires; the algorithm reaches line 10 and returns `true`.

If there is a reachable negative cycle, it returns `false`. This direction is a summing argument. Suppose the algorithm returned `true`. Then for every edge (u, v) with $d[u]$ finite,

$$d[v] \;\leq\; d[u] + w(u, v). \tag{13.12}$$

Let $c = \langle c_0, c_1, \ldots, c_k = c_0 \rangle$ be a reachable negative cycle, so $\sum_{i=1}^{k} w\,(c_{i-1}, c_i) < 0$. Every vertex on c is reachable from s, so every $d[c_i]$ is finite and Equation (13.12) applies to every edge of c. Sum Equation (13.12) around the cycle:

$$\sum_{i=1}^{k} d[c_i] \le \sum_{i=1}^{k} d[c_{i-1}] + \sum_{i=1}^{k} w(c_{i-1}, c_i).$$

Because the cycle closes, $c_0 = c_k$, so the two d-sums run over exactly the same multiset of vertices and are equal. Cancel them:

$$0 \le \sum_{i=1}^{k} w(c_{i-1}, c_i) = w(c) < 0,$$

which is absurd. So the algorithm cannot have returned `true`. ■

The same idea can be phrased through the δ_i notation of Equation (13.11), and it is worth having too: *if there is still some edge to relax in the graph, then there is a vertex u such that $\delta_n(s,u) < \delta_{n-1}(s,u)$. But there are only n vertices in G* — so a cheapest walk using n edges must repeat a vertex, hence contain a cycle, and since it is *cheaper* than anything with $n-1$ edges, that cycle must have negative weight.

> **WARNING.** `false` means "a negative cycle is reachable from s", not "the graph has a negative cycle". A negative cycle sitting in a component you cannot reach from s is invisible to this test and harmless to the answer. If you need to know whether *any* negative cycle exists anywhere, add a new vertex with a zero-weight edge to every other vertex and run Bellman–Ford from it — Exercise 13.14.

Nor does `false` tell you *where* the cycle is, only that one exists. Recovering it is easy once you know the trick: remember which vertex v fired the detector, walk back $|V|$ steps along π (which is guaranteed to land you *on* the cycle, since $|V|$ steps cannot stay off it), then walk π round until you return to that vertex. Exercise 13.15 asks you to implement this.

13.8.9 Running time

The whole count fits in one line:

$$T_{\mathrm{BF}} = \underbrace{\Theta(V)}_{\text{init}} + \underbrace{(|V|-1)\,|E|}_{\text{passes}} + \underbrace{|E|}_{\text{detector}} = \Theta(VE). \tag{13.13}$$

In detail, reading Equation (13.13) term by term: initialization is $\Theta(V)$. The main loop of lines 05–07 runs $|V|-1$ times, and each pass examines all $|E|$ edges at $\Theta(1)$ each, giving $\Theta(VE)$. The detector of lines 08–09 is one more $\Theta(E)$ sweep, absorbed. Total $\Theta(VE)$, and this is a Θ, not an O: without the early-exit optimization the algorithm always does all the work, whatever the graph looks like.

How bad is that? On a sparse graph with $E = \Theta(V)$ it is $\Theta(V^2)$ — the same as array-based Dijkstra, and worse than heap Dijkstra's $O(V\lg V)$. On a dense graph with $E = \Theta(V^2)$ it is $\Theta(V^3)$, which is the same as running Floyd–Warshall and getting *all pairs* for the same price. That is a good reason to reach for Chapter 15 when the graph is dense and has negative edges.

> **A caution about a popular optimization.** The early exit turns many practical instances into two or three passes, and a further refinement — keeping a queue of only those vertices whose d changed in the previous pass, known as SPFA or the Bellman–Ford–Moore algorithm — is often dramatically faster. But its worst case is still $\Theta(VE)$, and adversarial graphs that force

> the worst case are easy to construct and appear routinely in programming contests. Use it, but never quote a better bound than $\Theta(VE)$ for it.

13.9 Shortest Paths in a Directed Acyclic Graph

13.9.1 Why a DAG is easy

The whole idea fits in one sentence: *finding shortest paths in DAGs is much easier, because it is easy to find an order in which to do the relaxations — topological sorting!*

Recall §13.3.4: every algorithm in this chapter is an answer to the question *in what order should I relax edges, so that when I use $d[u]$ it is already final?* Dijkstra answers it with a priority queue and pays $O(\lg V)$ per operation for the privilege. Bellman–Ford refuses to answer it and pays a factor of $|V|$ instead. In a directed acyclic graph the answer is free: Chapter 11 showed how to compute a **topological order** — a listing of the vertices such that every edge goes forward — in $\Theta(V + E)$ time, by either depth-first finishing times or Kahn's in-degree method.

And a topological order is exactly what we need. If we process vertices in topological order, then by the time we reach u, every edge that could possibly have improved $d[u]$ has already been relaxed, because every such edge comes from a vertex earlier in the order. So $d[u]$ is already final, and we may relax the edges out of u once and never think about u again.

13.9.2 The algorithm

Figure 13.19 gives the pseudocode.

```
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.

Compare it with Dijkstra: line 04 replaces the priority queue entirely, and lines 05–07 replace the `while`-loop and its `Extract-Min`. Compare it with Bellman–Ford: the outer loop over $|V| - 1$ passes has vanished, because one pass in the right order does the work of all of them.

13.9.3 Why no priority queue is needed

THEOREM 13.14. If G is a directed acyclic graph, `DAG-Shortest-Paths(G,w,s)` terminates with $d[v] = \delta(s, v)$ for every $v \in V$, and G_π is a shortest-path tree rooted at s.

Proof. Let v be reachable from s and let $p = \langle v_0, v_1, \dots, v_k \rangle$, $v_0 = s$, $v_k = v$, be a shortest path from s to v. (A DAG has no cycles at all, so certainly no negative cycles, and Theorem 13.4 applies.)

Every edge (v_{i-1}, v_i) of p is an edge of G, so in a topological order v_{i-1} comes strictly before v_i. Therefore the vertices $v_0, v_1, \dots, v_k$ appear in the topological order in exactly that relative order. The algorithm processes vertices in topological order and, when it processes v_{i-1}, it relaxes (v_{i-1}, v_i). Hence the edges of p are relaxed in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ — possibly with many other relaxations in between, which the path-relaxation property permits. By Lemma 13.8, $d[v] = \delta(s, v)$ at the end, and Lemma 13.9 gives the tree. If v is unreachable, Lemma 13.6 gives $d[v] = \infty = \delta(s, v)$. ■

Notice what the proof did **not** need: no assumption about the sign of the weights. The DAG algorithm is perfectly happy with negative edges — a DAG cannot contain a cycle, so it cannot contain a negative cycle, and the pathology of §13.5.2 simply cannot arise. It is the only algorithm in this chapter that is both linear-time and negative-weight-safe, at the price of only working on acyclic graphs.

13.9.4 A worked example

We build the example from a graph the reader already knows: take the Bellman–Ford example graph of **Figure 13.14** and delete its three *backward* edges (x, t), (z, x) and (z, s). What is left, shown in **Figure 13.20**, is acyclic and keeps two of the three negative weights, $w(t, z) = -4$ and $w(y, x) = -3$; the third, $w(x, t) = -2$, was one of the edges deleted.

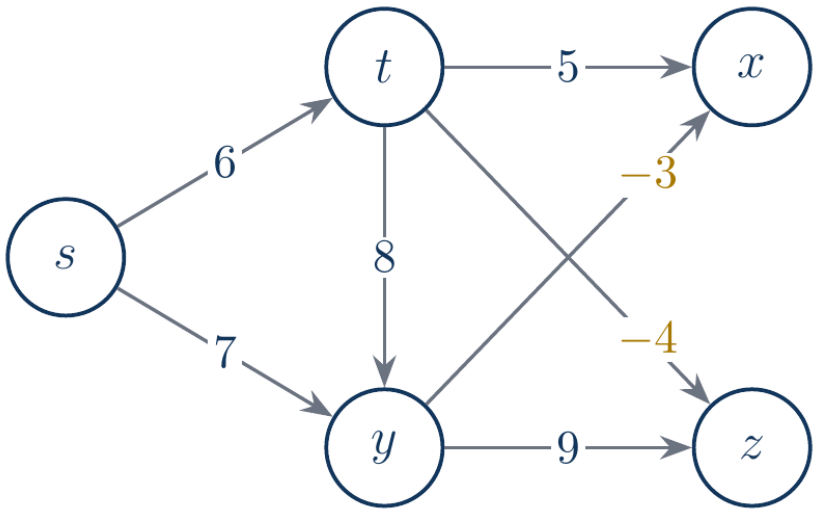


Figure 13.20 The Bellman–Ford example graph with its three backward edges removed: $s \to t$ (6), $s \to y$ (7), $t \to x$ (5), $t \to y$ (8), $t \to z$ (−4), $y \to x$ (−3), $y \to z$ (9). Every arrow now points left-to-right or downward, so the graph is a DAG — and it still contains two negative edges, which the DAG algorithm handles without complaint.

Step 1: topological sort. Kahn's algorithm from Chapter 11 computes in-degrees: s has 0, t has 1 (from s), y has 2 (from s and t), x has 2 (from t and y), z has 2 (from t and y). Only s is ready. Remove s: t drops to 0 and y drops to 1. Remove t: y drops to 0, x to 1, z to 1. Remove y: x and z drop to 0. Remove x, then z. The order is

$$s,\ t,\ y,\ x,\ z.$$

Step 2: one relaxation pass.

Take s, $d[s] = 0$. Relax (s,t): $0 + 6 = 6 < \infty$, so $d[t] \leftarrow 6$. Relax (s,y): $0 + 7 = 7 < \infty$, so $d[y] \leftarrow 7$.

Take t, $d[t] = 6$ *— and it is already final, because the only edge into t was* (s,t)*, already relaxed.* Relax (t,x): $6 + 5 = 11 < \infty$, so $d[x] \leftarrow 11$. Relax (t,y): $6 + 8 = 14$, not less than 7 — no change. Relax (t,z): $6 + (-4) = 2 < \infty$, so $d[z] \leftarrow 2$.

Take y, $d[y] = 7$*, final.* Relax (y,x): $7 + (-3) = 4 < 11$, so $d[x] \leftarrow 4$. Relax (y,z): $7 + 9 = 16$, not less than 2 — no change.

Take x, $d[x] = 4$*, final.* No outgoing edges left after the deletion.

Take z, $d[z] = 2$*, final.* No outgoing edges.

Done, in one pass. The answer is $\delta(s,s) = 0$, $\delta(s,t) = 6$, $\delta(s,y) = 7$, $\delta(s,x) = 4$, $\delta(s,z) = 2$. **Table 13.12** is the same run in table form.

Table 13.12 `DAG-Shortest-Paths` on **Figure 13.20**, one row per vertex in topological order. Compare with **Table 13.10**: Bellman–Ford needed four passes over ten edges — forty relaxations — to solve the full graph. Here seven relaxations suffice. Note also that the answers differ ($\delta(s,t)$ is 6 here and 2 there) because deleting (x,t) removed the cheap route; the two are different graphs, not two answers to the same question.

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

13.9.5 The program

Listing 13.5 `DAG-Shortest-Paths`: Kahn's topological sort followed by a single relaxation pass. The `if ((int)order.size() != n)` test is a free cycle detector — Kahn's algorithm cannot output every vertex if a cycle exists — so the program refuses to give an answer on a graph where its assumption fails. Compile with `g++ -std=c++17 -O2 -o dag dag.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 13.5 (Continued) *DAG-Shortest-Paths: Kahn's topological sort followed by a single relaxation pass. The if ((int)order.size() != n) test is a free cycle detector — Kahn's algorithm cannot output every vertex if a cycle exists — so the program refuses to give an answer on a graph where its assumption fails. 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
```

13.9.6 Running time

The answer is $\Theta(V + E)$ — *only one relaxation for each edge, a factor of about V faster than Bellman–Ford.*

The accounting: initialization $\Theta(V)$; topological sort $\Theta(V + E)$ by either method of Chapter 11; the main loop visits each vertex once and each edge once, $\Theta(V + E)$. Total $\Theta(V + E)$, which is **linear in the size of the input** — you cannot do better, because you must at least read the graph. There is no lg factor because there is no priority queue, and no factor of $|V|$ because there is no repetition.

> **KEY IDEA.** The topological order does for free what the priority queue does at a cost of $O(\lg V)$ per vertex: it tells you which vertex is safe to finalize next. When the structure of the graph hands you that information, do not pay for a data structure to rediscover it.

13.9.7 Longest paths and the critical path

A bonus that falls out for nothing. In a general graph the *longest simple path* problem is NP-hard — §13.3.3 explained why the cut-and-paste argument breaks. In a DAG it is trivial: negate every weight, run `DAG-Shortest-Paths`, negate the answer. Equivalently, replace `<` with `>` and ∞ with $-\infty$ in the relaxation and run the same loop.

This is the **critical path method** of project scheduling. Vertices are milestones, an edge (u, v) of weight w means "v cannot start until w time units after u", and the longest path from "project start" to "project end" is the shortest possible duration of the whole project. Every edge on that path is *critical*: delay any one of them and the whole project slips. Exercise 13.11 asks you to compute one for the Profitina Laundry pipeline.

13.10 Reconstructing the Path, Not Just Its Length

Everything so far has computed numbers. Nobody wants numbers. A courier wants a route; a router wants a next hop; a project manager wants a list of tasks. This section turns π into that.

13.10.1 The predecessor array is the answer

Lemma 13.9 did the theoretical work: when every $d[v] = \delta(s, v)$, the predecessor subgraph G_π is a shortest-path tree, and the unique path from s to v inside it is a shortest path in G. All that remains is to read it.

The reading is backward. $\pi[v]$ is the *last* vertex before v; $\pi[\pi[v]]$ is the one before that; and so on until we reach s, whose π is NIL. So we collect vertices from v back to s and then reverse the list.

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 the path in the right order without an explicit reversal: line 6 prints everything before v, then line 7 prints v. Its depth is the number of edges on the path, at most $|V| - 1$, so on a large graph an iterative version with an explicit vector is safer. **Listing 13.6** gives both.

13.10.2 The program

Listing 13.6 Path reconstruction from π, iteratively and recursively, with two safety features: a cycle guard in `buildPath`, and a re-computation of each path's weight from the edge list to check it against d. The graph is the Dijkstra example graph plus an extra vertex q that nothing points at.

```
#include <iostream>
#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; };

// PRINT-PATH(G, s, v) from the predecessor array, written without recursion.
// Returns false when v is unreachable from s.
bool buildPath(const vector<int>& pi, int s, int v, vector<int>& path) {
    path.clear();
    int guard = 0, n = (int)pi.size();
    for (int t = v; t != -1; t = pi[t]) {
        path.push_back(t);
        if (++guard > n) return false;        // pi has a cycle: refuse to loop for ever
    }
    reverse(path.begin(), path.end());
    return !path.empty() && path.front() == s;
}

// The recursive form, exactly as the textbook writes it.
void printPathRec(const vector<string>& name, const vector<int>& pi, int s, int v) {
    if (v == s)            { cout << name[s]; return; }
    if (pi[v] == -1)       { cout << "no path from " << name[s] << " to " << name[v];
                             return; }
    printPathRec(name, pi, s, pi[v]);
    cout << " -> " << name[v];
}

int main() {
    // Same graph as the Dijkstra listing, plus an isolated vertex q that
    // nothing points at, so we can see what "no path" looks like.
    vector<string> name = {"s", "u", "v", "x", "y", "q"};
    int n = (int)name.size();
    auto id = [&](const string& t) {
        return (int)(find(name.begin(), name.end(), t) - 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});
    };
```

Listing 13.6 (Continued) *Path reconstruction from \pi, iteratively and recursively, with two safety features: a cycle guard in buildPath, and a re-computation of each path's weight from the edge list to check it against d. The graph is the Dijkstra example graph plus an extra vertex q that nothing points at.*

```
    E("s","u",10); E("s","x", 5);
    E("u","v", 1); E("u","x", 2);
    E("x","u", 3); E("x","v", 9); E("x","y", 2);
    E("v","y", 4);
    E("y","v", 6); E("y","s", 7);

    int s = id("s");
    vector<long long> d(n, INF);
    vector<int> pi(n, -1);
    d[s] = 0;
    priority_queue<pair<long long,int>, vector<pair<long long,int>>,
                   greater<pair<long long,int>>> Q;
    Q.push({0, s});
    while (!Q.empty()) {
        auto [key, u] = Q.top(); Q.pop();
        if (key > d[u]) continue;
        for (const Edge& e : adj[u])
            if (d[u] + e.w < d[e.to]) {
                d[e.to] = d[u] + e.w; pi[e.to] = u; Q.push({d[e.to], e.to});
            }
    }

    for (int v = 0; v < n; ++v) {
        cout << "s to " << name[v] << ": ";
        vector<int> path;
        if (!buildPath(pi, s, v, path)) { cout << "unreachable\n"; continue; }
        long long check = 0;
        for (size_t i = 0; i + 1 < path.size(); ++i) {       // re-add the weights
            for (const Edge& e : adj[path[i]])
                if (e.to == path[i+1]) { check += e.w; break; }
        }
        for (size_t i = 0; i < path.size(); ++i) {
            if (i) cout << " -> ";
            cout << name[path[i]];
        }
        cout << "   weight " << check << " (d = " << d[v] << ")"
             << (check == d[v] ? "  OK" : "  MISMATCH") << "\n";
    }

    cout << "\nthe recursive PRINT-PATH, for comparison:\n";
    for (int v = 0; v < n; ++v) { cout << "  "; printPathRec(name, pi, s, v); cout << "\n";
}
    return 0;
}
```

```
s to s: s   weight 0 (d = 0)  OK
s to u: s -> x -> u   weight 8 (d = 8)  OK
s to v: s -> x -> u -> v   weight 9 (d = 9)  OK
s to x: s -> x   weight 5 (d = 5)  OK
s to y: s -> x -> y   weight 7 (d = 7)  OK
s to q: unreachable

the recursive PRINT-PATH, for comparison:
  s
  s -> x -> u
  s -> x -> u -> v
  s -> x
  s -> x -> y
  no path from s to q
```

Every reported path weighs exactly what d says it should — which is not a tautology but a genuine check, since the weights are recomputed from the adjacency lists and compared against a number produced by a completely different piece of code.

13.10.3 Three traps

Reading π before the algorithm has finished. Half-way through Dijkstra, $\pi[v]$ points at the best predecessor *found so far*. In the trace of §13.7.4, $\pi[v]$ was x, then y, then u. Walking π at step 3 would produce a real path, of weight 13, that is simply not shortest. Only when the algorithm has terminated is Lemma 13.9 available.

Forgetting to set π at all. The Dijkstra pseudocode of **Figure 13.4** updates only d. If you transcribe it literally you will get correct distances and an array of NILs. Always set $\pi[v] \leftarrow u$ in the same `if` that sets $d[v]$ — never outside it, or you will record a predecessor for an improvement that did not happen.

Following π after Bellman–Ford returned `false`. With a reachable negative cycle, π can contain a cycle, and a naive backward walk loops for ever. The guard in `buildPath` — counting steps and giving up after $|V|$ of them — costs nothing and turns an infinite loop into a clean answer. Use it in any code that might see untrusted weights.

13.11 Which Algorithm for Which Graph

Table 13.13 is the whole chapter compressed into a decision aid; the prose beneath it is the same thing said as a procedure.

Table 13.13 Which single-source algorithm to use. Read it top-down: first ask whether the graph is acyclic, then whether any weight is negative, then how dense it is. 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 decision procedure in words, and it is worth committing to memory:

1. **Are all edge weights equal?** Use breadth-first search. $\Theta(V + E)$ and no arithmetic at all.
2. **Is the graph acyclic?** Use `DAG-Shortest-Paths`. $\Theta(V + E)$, and negative weights are fine.
3. **Are all weights non-negative?** Use Dijkstra. Binary heap if sparse, plain array if dense.
4. **Otherwise** use Bellman–Ford, and *check its return value* before you use its output.
5. **Do you need all pairs?** Go to Chapter 15.

> **EXAMPLE (three decisions).** *A road network of 400,000 junctions and 1,000,000 roads, weights being travel minutes.* Cyclic, non-negative, very sparse — Dijkstra with a binary heap. *A build-dependency graph of 5,000 tasks with durations.* Acyclic — `DAG-Shortest-Paths`, or its longest-path twin for the critical path. *A table of 40 currencies with all pairwise exchange rates, weights* −ln(rate)*.* Cyclic, dense, and negative weights are the entire point — Bellman–Ford, run precisely to see whether it returns `false`.

13.12 Profitina in Practice

13.12.1 Where shortest paths live inside an AI-visibility platform

Profitina is an AI-visibility (GEO) platform: it scans a website, scores how ready that site is to be *cited* by an AI answer engine, and explains each signal it measured [9]. Three parts of that product are shortest-path problems wearing different clothes.

The crawl frontier is a priority queue keyed by a running cost. A scan does not fetch a site's pages in the order it discovers them; it fetches them in order of a cost that accumulates along the link path from the entry URL — click depth, redirect hops, robots-delay budget, the time already spent on this host. That is precisely $d[u] + w(u, v)$, and the frontier that always pops the cheapest next page is precisely Dijkstra's priority queue. All the weights involved are non-negative — no fetch ever gives time back — so Dijkstra is the right algorithm and its $O(E\lg V)$ bound is the crawl's cost model. Chapter 11 observed that the same queue serves Prim; §13.7.11 explains why the two are not interchangeable here: the crawl budget is a sum along a route, not the cost of one hop.

The scan pipeline is a DAG, and its schedule is a longest path. Chapter 11 showed that the stages of a scan — fetch, extract, parse structured data, evaluate each signal, aggregate, store, notify — form a directed acyclic graph of dependencies. Weight each edge by the stage's duration and the *longest* path through that DAG is the minimum time a full scan can take, by the critical-path argument of §13.9.7. It is computed in $\Theta(V + E)$ with no priority queue, because topological order is already known.

Weekly score trends and re-scan cadence. The historical analytics dashboard re-scans each brand at a per-tier cadence. Choosing which brand to scan next, given per-brand costs and staleness penalties, is again a cheapest-next-step decision over a graph whose weights are non-negative.

13.12.2 The worked case study: Profitina Laundry

Profitina Laundry is the teaching case study of this book: a small laundry in Surabaya with a van, a depot, and a handful of collection points. This morning's questions are the three of this chapter.

Question 1. What is the fastest route from the depot to each collection point? The graph has seven vertices — Depot, Kertajaya, Dharmahusada, Manyar, ITS, Mulyosari, Keputih — and eleven one-way legs, each weighted by driving minutes. Minutes are never negative, so this is Dijkstra.

Question 2. What does each route *cost*, when one stop pays a rebate? The campus partner at ITS pays the laundry to collect there, so the leg (ITS,Keputih) has a *negative* net cost of −15 thousand rupiah. Dijkstra may no longer be used — §13.7.3 — so this is Bellman–Ford.

Question 3. What happens if the rebate is set too generously? If the rebate exceeds the cost of the loop that returns to it, a driver could circle for ever and be paid for it. That is a negative cycle,

and detecting it is the difference between a tariff sheet and a bankruptcy. **Listing 13.7** answers all three questions in one program.

Listing 13.7 The Profitina Laundry case study: Dijkstra on driving minutes, Bellman–Ford on net cost with one negative leg, and the negative-cycle detector firing when the rebate is raised too far. Compile with `g++ -std=c++17 -O2 -o laundry laundry.cpp`.

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <map>
#include <algorithm>
#include <climits>

using namespace std;

const long long INF = LLONG_MAX / 4;

struct Leg { int u, v; long long w; };

struct Net {                        // a tiny directed weighted graph with named vertices
    vector<string> name;
    map<string,int> idx;
    vector<vector<pair<int,long long>>> adj;
    vector<Leg> edges;                          // the same edges as a flat list

    int id(const string& s) {
        auto it = idx.find(s);
        if (it != idx.end()) return it->second;
        idx[s] = (int)name.size();
        name.push_back(s);
        adj.push_back({});
        return (int)name.size() - 1;
    }
    void add(const string& a, const string& b, long long w) {
        int u = id(a), v = id(b);
        adj[u].push_back({v, w});
        edges.push_back({u, v, w});
    }
    int n() const { return (int)name.size(); }
};

string pathOf(const Net& g, const vector<int>& pi, int v) {
    vector<string> p;
    for (int t = v; t != -1; t = pi[t]) p.push_back(g.name[t]);
    reverse(p.begin(), p.end());
    string out;
    for (size_t i = 0; i < p.size(); ++i) { if (i) out += " -> "; out += p[i]; }
    return out;
}
```

Listing 13.7 (Continued) *The Profitina Laundry case study: Dijkstra on driving minutes, Bellman–Ford on net cost with one negative leg, and the negative-cycle detector firing when the rebate is raised too far. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
void dijkstra(const Net& g, int s, vector<long long>& d, vector<int>& pi) {
    d.assign(g.n(), INF);
    pi.assign(g.n(), -1);
    d[s] = 0;
    priority_queue<pair<long long,int>, vector<pair<long long,int>>,
                   greater<pair<long long,int>>> Q;
    Q.push({0, s});
    while (!Q.empty()) {
        auto [key, u] = Q.top(); Q.pop();
        if (key > d[u]) continue;
        for (auto [v, w] : g.adj[u])
            if (d[u] + w < d[v]) { d[v] = d[u] + w; pi[v] = u; Q.push({d[v], v}); }
    }
}

bool bellmanFord(const Net& g, int s, vector<long long>& d, vector<int>& pi) {
    d.assign(g.n(), INF);
    pi.assign(g.n(), -1);
    d[s] = 0;
    for (int i = 1; i <= g.n() - 1; ++i)
        for (const Leg& e : g.edges)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) { d[e.v] = d[e.u] + e.w; pi[e.v] =
e.u; }
    for (const Leg& e : g.edges)                 // the extra pass: the detector
        if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) return false;
    return true;
}

int main() {
    cout << "PROFITINA LAUNDRY - morning pickup round\n";
    cout << "========================================\n\n";

    // ---- part 1: driving minutes between stops. All non-negative: Dijkstra. ----
    Net road;
    road.add("Depot",       "Kertajaya",   7);
    road.add("Depot",       "Dharmahusada",11);
    road.add("Kertajaya",   "Manyar",       4);
    road.add("Kertajaya",   "Dharmahusada", 3);
    road.add("Dharmahusada","ITS",          9);
    road.add("Manyar",      "ITS",          6);
    road.add("Manyar",      "Mulyosari",   10);
    road.add("ITS",         "Keputih",      5);
    road.add("ITS",         "Mulyosari",    4);
    road.add("Mulyosari",   "Keputih",      8);
    road.add("Keputih",     "Depot",       16);
```

Listing 13.7 (Continued) *The Profitina Laundry case study: Dijkstra on driving minutes, Bellman–Ford on net cost with one negative leg, and the negative-cycle detector firing when the rebate is raised too far. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
    vector<long long> d; vector<int> pi;
    dijkstra(road, road.id("Depot"), d, pi);

    cout << "minutes from the depot, and the road to take:\n";
    cout << left << setw(15) << "stop" << right << setw(5) << "min" << "   route\n";
    for (int v = 0; v < road.n(); ++v)
        cout << left << setw(15) << road.name[v] << right << setw(5) << d[v]
             << "   " << pathOf(road, pi, v) << "\n";

    // ---- part 2: net cost in rupiah-thousands. A partner drop-point pays a
    //      rebate, so one edge is negative. Dijkstra may not be used here. ----
    cout << "\nnet cost per route leg (negative = partner rebate):\n";
    Net cost;
    cost.add("Depot",       "Kertajaya",   14);
    cost.add("Depot",       "Dharmahusada",22);
    cost.add("Kertajaya",   "Manyar",       8);
    cost.add("Kertajaya",   "Dharmahusada", 6);
    cost.add("Dharmahusada","ITS",         18);
    cost.add("Manyar",      "ITS",         12);
    cost.add("Manyar",      "Mulyosari",   20);
    cost.add("ITS",         "Keputih",    -15);   // campus partner pays a rebate
    cost.add("ITS",         "Mulyosari",    8);
    cost.add("Mulyosari",   "Keputih",     16);
    cost.add("Keputih",     "Depot",       30);

    vector<long long> dc; vector<int> pic;
    bool ok = bellmanFord(cost, cost.id("Depot"), dc, pic);
    cout << "BELLMAN-FORD returns " << (ok ? "true" : "false") << "\n";
    cout << left << setw(15) << "stop" << right << setw(6) << "cost" << "   route\n";
    for (int v = 0; v < cost.n(); ++v)
        cout << left << setw(15) << cost.name[v] << right << setw(6) << dc[v]
             << "   " << pathOf(cost, pic, v) << "\n";

    // ---- part 3: the same tariff sheet with a rebate that is too generous ----
    cout << "\nnow suppose the rebate is raised to -80:\n";
    for (Leg& e : cost.edges)
        if (cost.name[e.u] == "ITS" && cost.name[e.v] == "Keputih") e.w = -80;
    ok = bellmanFord(cost, cost.id("Depot"), dc, pic);
    cout << "BELLMAN-FORD returns " << (ok ? "true" : "false")
         << "  (a driver could circle for ever and be paid for it)\n";
    return 0;
}
```

```
PROFITINA LAUNDRY - morning pickup round
========================================

minutes from the depot, and the road to take:
stop             min   route
Depot              0   Depot
Kertajaya          7   Depot -> Kertajaya
Dharmahusada      10   Depot -> Kertajaya -> Dharmahusada
Manyar            11   Depot -> Kertajaya -> Manyar
ITS               17   Depot -> Kertajaya -> Manyar -> ITS
Mulyosari         21   Depot -> Kertajaya -> Manyar -> Mulyosari
Keputih           22   Depot -> Kertajaya -> Manyar -> ITS -> Keputih

net cost per route leg (negative = partner rebate):
BELLMAN-FORD returns true
stop             cost   route
Depot               0   Depot
Kertajaya          14   Depot -> Kertajaya
Dharmahusada       20   Depot -> Kertajaya -> Dharmahusada
Manyar             22   Depot -> Kertajaya -> Manyar
ITS                34   Depot -> Kertajaya -> Manyar -> ITS
Mulyosari          42   Depot -> Kertajaya -> Manyar -> Mulyosari
Keputih            19   Depot -> Kertajaya -> Manyar -> ITS -> Keputih

now suppose the rebate is raised to -80:
BELLMAN-FORD returns false  (a driver could circle for ever and be paid for it)
```

Three things are worth reading out of that output.

The direct leg Depot → Dharmahusada takes 11 minutes, but going the long way round through Kertajaya takes $7 + 3 = 10$. The shortest route is not the one with fewest legs — the lesson of §13.2.2, in a van.

Keputih costs 19 to reach but ITS, which is *on the way to it*, costs 34. A cost that falls as you go further is exactly what a negative edge produces, and exactly what breaks Dijkstra's guarantee: Corollary 13.11 promises that the extracted keys never decrease, yet here a vertex at cost 34 leads to one at cost 19. On this particular graph a lazy-deletion Dijkstra would still happen to print the right numbers, because Keputih cannot be reached at all before ITS is extracted — but that is luck, not a theorem. On any graph where a vertex is finalized before its cheap, negative in-edge is examined, the answer becomes wrong in exactly the way **Figure 13.5** shows. Bellman–Ford needs no luck.

And when the rebate is raised from -15 to -80, the loop Depot → Kertajaya → Manyar → ITS → Keputih → Depot costs $14 + 8 + 12 - 80 + 30 = -16$, and the detector fires. No amount of cleverness in the routing code would have caught that; only the extra pass of lines 08–09 does.

13.13 Common Mistakes, and How to Avoid Them

1. Using Dijkstra on a graph with a negative edge. The commonest error in the whole subject, because the program does not complain. Add an assertion: if any weight is negative, refuse to run Dijkstra. See **Figure 13.5**.

2. Believing that adding a constant to every weight fixes negative edges. It does not, because a k-edge path gains kc while a 2-edge path gains only $2c$, so the comparison between them changes. Exercise 13.6.

3. Confusing a negative edge with a negative cycle. A negative edge is ordinary; Bellman–Ford and the DAG algorithm handle it exactly as they handle any other. Only a *cycle* of negative total weight destroys the problem. **Figure 13.2**.

4. Running Bellman–Ford for $|V|$ passes instead of $|V| - 1$, or for $|E|$ passes. The first wastes a pass; the second wastes many. Theorem 13.4 fixes the number at $|V| - 1$ because a simple path has at most that many edges.

5. Omitting the detector pass. Without lines 08–09 you get numbers with no indication of whether they mean anything. The pass costs $\Theta(E)$ — one part in $|V|$ of the total — and is never worth skipping.

6. Relaxing out of an unreached vertex. `INF + (-4)` is less than `INF`, so without the `d[u] < INF` guard the estimate of an unreachable vertex silently becomes finite. Lemma 13.6 in code.

7. Signed overflow from a too-large sentinel. Using `INT_MAX` as ∞ and then computing `d[u] + w` wraps to a large negative number, which looks like a wonderful shortcut to the relaxation test. Use `LLONG_MAX/4`, or guard first and add second — this chapter's programs do both.

8. Updating d without updating π. You get lengths and no routes. §13.10.3.

9. Reading π before termination. A half-built predecessor array describes a real path that is not yet shortest.

10. Forgetting `DecreaseKey`, or faking it wrongly. If $d[v]$ falls and the queue is not told, the queue will hand back vertices in the wrong order and the answer will be wrong. Either implement `DecreaseKey` properly or use the lazy-deletion trick of **Listing 13.2** — pushing a second copy and discarding stale ones on extraction. What you must not do is push a second copy and *fail* to discard the stale one, which makes the algorithm process a vertex twice.

11. Assuming the shortest-path tree is the minimum spanning tree. They are different objects answering different questions. §13.1.2 and Exercises 13.4 and 13.5.

12. Running the DAG algorithm on a graph that is not a DAG. It returns an answer, and the answer is wrong. Kahn's topological sort detects the cycle for free — **Listing 13.5** checks `order.size() != n` and refuses.

13. Quoting an improved bound for SPFA. The queue-based refinement of Bellman–Ford is fast on typical input and $\Theta(VE)$ in the worst case. Never write $O(E)$ next to it.

14. Treating $\delta(s, v) = \infty$ as an error. It is an answer: there is no route. Print "unreachable" and move on.

13.14 Chapter Summary

The **shortest-path problem** in a weighted digraph $G = (V, E, w)$ asks for $\delta(s, v)$, the minimum weight of any path from s to v, where the weight of a path is $w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$. Of the four variants — single-pair, single-source, single-destination, all-pairs — we solve **single-source**, because the first reduces to it with no known saving, the third is it on the transpose graph, and the fourth is either $|V|$ runs of it or the subject of Chapter 15.

Two structural facts underlie everything. **Optimal substructure** (Theorem 13.1): every subpath of a shortest path is itself a shortest path, proved by cutting out a supposedly better subpath and pasting it in. **The triangle inequality** (Theorem 13.2, Corollary 13.3): $\delta(s, v) \leq \delta(s, u) + w(u, v)$ for every edge, because the route through u is one candidate among all routes to v.

Negative edges are harmless; negative cycles are fatal. A reachable cycle of negative total weight makes $\delta(s, v) = -\infty$ for everything downstream of it, because each extra lap is cheaper. Where no such cycle exists, every shortest path may be taken simple and therefore has at most $|V| - 1$ edges (Theorem 13.4) — the fact that fixes Bellman–Ford's pass count.

Relaxation is the only primitive: if $d[v] > d[u] + w(u, v)$, set $d[v] \leftarrow d[u] + w(u, v)$ and $\pi[v] \leftarrow u$. Five properties make it safe, and every correctness proof in the chapter reduces to them: the **upper-bound** property ($d \geq \delta$ always, and frozen once equal), the **no-path** property, the **convergence** property (right answer for u plus one relaxation gives the right answer for v), the **path-relaxation** property (relax a shortest path's edges in order and its endpoint is solved), and the **predecessor-subgraph** property (G_π is a shortest-path tree when all $d = \delta$).

Dijkstra finalizes vertices in increasing order of distance, using a priority queue keyed by d. It is correct only for $w \geq 0$; the proof's Step 5 is where that assumption is spent, and **Figure 13.5** is a three-vertex graph where dropping it gives a wrong answer with no warning. Its cost is $|V|$ `ExtractMin` plus $|E|$ `DecreaseKey`: $\Theta(V^2)$ with an array, $O(E\lg V)$ with a binary heap, $O(V\lg V + E)$ with a Fibonacci heap.

Bellman–Ford makes no choices at all: it relaxes every edge $|V| - 1$ times, which the path-relaxation property shows is enough, and then relaxes every edge once more. If anything still improves, a negative cycle is reachable and it returns `false`. Cost $\Theta(VE)$.

On a DAG, topological order tells us for free what the priority queue charges $O(\lg V)$ to discover, so one relaxation per edge in that order solves the problem in $\Theta(V + E)$, negative weights included. Reversing the comparison gives longest paths and the critical path of a project.

Finally, the answer anybody actually wanted is the **route**, recovered by walking π backward from v to s and reversing — valid, by Lemma 13.9, only once the algorithm has terminated.

13.14.1 Looking ahead

Chapter 14 steps away from graphs to **amortized analysis**, the technique that justifies the "amortized $\Theta(1)$" in the Fibonacci-heap row of **Table 13.8** — a claim we have used twice now without proving. Chapter 15 returns to shortest paths and asks for **all pairs** at once, with Floyd–Warshall's $\Theta(V^3)$ dynamic program and with Johnson's algorithm, whose entire purpose is to *reweight* a graph with negative edges so that Dijkstra — fast, and forbidden here — becomes legal.

13.15 Exercises

13.1 Compute $w(p)$ for $p = \langle s, x, y, v \rangle$ in the Dijkstra example graph, and say whether p is a shortest path from s to v.

13.2 Give an example of a weighted digraph with two distinct shortest paths between the same pair of vertices.

13.3 Prove that if all edge weights are positive (strictly), then every shortest path is simple.

13.4 Give a connected undirected weighted graph on four vertices in which the shortest-path tree rooted at some vertex s is not a minimum spanning tree.

13.5 Prove or disprove: the total weight of a shortest-path tree rooted at s is at least the total weight of a minimum spanning tree of the same graph.

13.6 A student proposes to handle negative weights by adding a constant $c = |\min_e w(e)|$ to every edge and then running Dijkstra. Give a graph on which this returns the wrong shortest path, and explain the flaw in one sentence.

13.7 Run Dijkstra by hand on the Dijkstra example graph with source x instead of s. Give the trace table in the form of **Table 13.6**.

13.8 In the Bellman–Ford example graph, what is the smallest number of passes that would still produce the correct answer if the edge list were reordered optimally? Give such an order.

13.9 Prove that in Dijkstra, once a vertex u has been extracted, no subsequent relaxation can change $d[u]$ — assuming $w \geq 0$.

13.10 Complete the proof of Lemma 13.9 for the case of a zero-weight cycle in G_π: show that such a cycle cannot arise, given that $\pi[s]$ is never assigned.

13.11 The Profitina Laundry processing pipeline is: *receive* → *sort* (10 min), *sort* → *wash* (35), *sort* → *dry-clean* (50), *wash* → *dry* (40), *dry-clean* → *press* (15), *dry* → *press* (15), *press* → *fold* (10), *fold* → *dispatch* (5). Draw the DAG, find a topological order, and compute the critical path from *receive* to *dispatch*.

13.12 Modify Dijkstra so that, among all shortest paths, it returns one with the fewest edges. Prove your modification correct.

13.13 Modify Bellman–Ford so that instead of returning `false` it labels every vertex with $+\infty$, a finite value, or $-\infty$, correctly. Give the algorithm and its running time.

13.14 Give an $O(VE)$ algorithm that decides whether a digraph contains *any* negative cycle, reachable from a given source or not.

13.15 Extend Bellman–Ford so that when it detects a negative cycle it also prints the vertices of one such cycle, in order. Justify the $|V|$-step walk your method uses.

13.16 Let G have non-negative weights and let s be a source. Prove that if $\delta(s,u) = \delta(s,v)$ and $(u,v) \in E$ then $w(u,v) \geq 0$ — and give an example showing the converse fails.

13.16 Solutions to the Exercises

13.1 $w(p) = w(s,x) + w(x,y) + w(y,v) = 5 + 2 + 6 = 13$. It is *not* shortest: $\delta(s,v) = 9$, achieved by $\langle s,x,u,v\rangle$ — see **Table 13.1**.

13.2 Two vertices s and t joined by two parallel routes of equal total weight: $s \to a \to t$ with $w(s,a) = 1$, $w(a,t) = 1$, and $s \to b \to t$ with $w(s,b) = 1$, $w(b,t) = 1$. Both weigh 2 and both are shortest. Every algorithm in this chapter keeps whichever of the two it discovers *first*, because `RELAX` replaces a route only on a strict improvement; neither is more correct.

13.3 Let p be a shortest path from s to v and suppose it repeats a vertex, so it contains a cycle c with at least one edge. All weights are strictly positive, so $w(c) > 0$. Excising c leaves a path p' with $w(p') = w(p) - w(c) < w(p)$, contradicting the minimality of p. Hence p repeats no vertex. (With weights merely non-negative the conclusion weakens to "*some* shortest path is simple" — Theorem 13.4 — because a zero-weight cycle can be present without penalty.)

13.4 Vertices s,a,b,c; edges $\{s,a\} = 1$, $\{s,b\} = 1$, $\{s,c\} = 1$, $\{a,b\} = 1$. The minimum spanning tree may use $\{s,a\},\{a,b\},\{s,c\}$, total weight 3. The shortest-path tree from s uses $\{s,a\},\{s,b\},\{s,c\}$, total weight 3 — equal here, so strengthen: make $\{a,b\} = 0.5$. Now the MST is

$\{s,a\},\{a,b\},\{s,c\}$ of weight 2.5, while the shortest-path tree from s is still $\{s,a\},\{s,b\},\{s,c\}$ of weight 3, since $\delta(s,b)=1$ by the direct edge and 1.5 through a. The trees differ, and the shortest-path tree is heavier.

13.5 True, and Exercise 13.4 shows the inequality can be strict. Proof: a shortest-path tree rooted at s is a spanning tree of the reachable component (Lemma 13.9), and a minimum spanning tree is by definition a spanning tree of minimum total weight. Hence the MST's total weight is at most that of any spanning tree, in particular the shortest-path tree's. The two optimize different objectives: the MST minimizes the *sum* of chosen edge weights, a shortest-path tree minimizes each *root-to-vertex distance* independently.

13.6 Take three vertices s, a, t and three edges: $w(s,a)=-2$, $w(a,t)=-2$ and $w(s,t)=-3$. In the original graph the two-edge route $s\to a\to t$ weighs -4 and the direct edge weighs -3, so $\delta(s,t)=-4$ via a. The most negative weight is -3, so $c=3$, and after adding it every weight is non-negative: $w(s,a)=1$, $w(a,t)=1$, $w(s,t)=0$. Now the two-edge route weighs 2 and the direct edge weighs 0, so Dijkstra on the shifted graph correctly finds the *shifted* shortest path, the direct edge — which is the wrong path for the original graph. The flaw, in one sentence: a k-edge path is inflated by kc, so shifting penalizes paths in proportion to their number of edges and can change which path is cheapest.

13.7 From source x: $d[x]=0$ initially. Step 1 extracts x (0) and relaxes $(x,u)\to 3$, $(x,v)\to 9$, $(x,y)\to 2$; $Q=\{y=2,u=3,v=9,s=\infty\}$. Step 2 extracts y (2) and relaxes (y,v): $2+6=8<9$ so $d[v]\leftarrow 8$; (y,s): $2+7=9<\infty$ so $d[s]\leftarrow 9$; $Q=\{u=3,v=8,s=9\}$. Step 3 extracts u (3) and relaxes (u,v): $3+1=4<8$ so $d[v]\leftarrow 4$; (u,x): no change; $Q=\{v=4,s=9\}$. Step 4 extracts v (4), relaxes (v,y): $4+4=8>2$, no change; $Q=\{s=9\}$. Step 5 extracts s (9), relaxes (s,u) and (s,x): no change. Final: $\delta(x,x)=0$, $\delta(x,y)=2$, $\delta(x,u)=3$, $\delta(x,v)=4$, $\delta(x,s)=9$.

13.8 One pass is enough. The shortest-path tree here is the single path $s\to y\to x\to t\to z$. List its four edges first, in that order — $(s,y),(y,x),(x,t),(t,z)$ — followed by the remaining six edges in any order. During that single pass the four tree edges are relaxed in path order, so the path-relaxation property (Lemma 13.8) is satisfied within the pass and every d reaches its true value. Zero passes cannot work, since $d[z]=\infty$ before any relaxation. The detector sweep of lines 08–09 still runs afterward, as it always must, to confirm that nothing more can improve.

13.9 Suppose u is extracted at some moment with $d[u]=\delta(s,u)$ (Theorem 13.10). By the upper-bound property, $d[u]\geq\delta(s,u)$ holds always, and a relaxation only ever *decreases* $d[u]$. A decrease would give $d[u]<\delta(s,u)$, contradicting the upper-bound property. Hence no relaxation after extraction can change $d[u]$. Note that the argument needs $w\geq 0$ only through Theorem 13.10.

13.10 Suppose G_π contains a cycle $\langle c_0,\dots,c_k=c_0\rangle$. Every vertex on it has π assigned, so none of them is s — because $\pi[s]$ is never assigned by any line of any of our algorithms, s has in-degree 0 in G_π and cannot lie on a cycle. Each c_i therefore had $\pi[c_i]$ set by a relaxation of (c_{i-1},c_i), at which moment $d[c_i]=d[c_{i-1}]+w(c_{i-1},c_i)$; since $d[c_{i-1}]$ can only have fallen afterward, presently $d[c_i]\geq d[c_{i-1}]+w(c_{i-1},c_i)$. Summing round the cycle and canceling the d terms gives $0\geq w(\text{cycle})$. If $w(\text{cycle})<0$ we contradict the hypothesis of no negative cycle. If $w(\text{cycle})=0$, consider the *last* of the k assignments to be made; at that moment the corresponding inequality was strict, because the relaxation's test requires $d[c_i]>d[c_{i-1}]+w(c_{i-1},c_i)$ *before* the assignment and no later change touched $d[c_{i-1}]$. A strict inequality anywhere in the sum gives $0>w(\text{cycle})=0$, absurd. So G_π is acyclic.

13.11 Topological order: *receive, sort, wash, dry-clean, dry, press, fold, dispatch* (also valid: swapping *dry-clean* earlier). Longest-path relaxation from *receive*: $\ell[\text{sort}] = 10$; $\ell[\text{wash}] = 45$; $\ell[\text{dry-clean}] = 60$; $\ell[\text{dry}] = 85$; $\ell[\text{press}] = \max(60 + 15, 85 + 15) = 100$; $\ell[\text{fold}] = 110$; $\ell[\text{dispatch}] = 115$. The critical path is *receive* → *sort* → *wash* → *dry* → *press* → *fold* → *dispatch*, 115 minutes. Dry-cleaning has $85 - 60 = 25$ minutes of slack: it may start late or run long by up to 25 minutes without delaying dispatch.

13.12 Keep a second array $h[v]$, the number of edges on the recorded route. Initialize $h[s] = 0$, $h[v] = \infty$. Change the relaxation to: if $d[u] + w(u, v) < d[v]$, or ($d[u] + w(u, v) = d[v]$ **and** $h[u] + 1 < h[v]$), then set $d[v] \leftarrow d[u] + w(u, v)$, $h[v] \leftarrow h[u] + 1$, $\pi[v] \leftarrow u$. The priority queue must also be keyed by the *pair* $(d[v], h[v])$, compared lexicographically — first by d, and by h only to break ties — so that among vertices at the same distance the one reached with fewer edges is extracted first. *Correctness:* think of every edge as carrying the pair weight $(w(u, v),1)$ and add pairs component by component. Pairs compared lexicographically behave exactly like numbers for everything §13.6 and Theorem 13.10 use: adding a pair with non-negative first component and positive second component never makes a route lexicographically smaller, so the one place where the proof spends $w \geq 0$ (Step 5) still goes through. Dijkstra on pair weights therefore finds, for every v, the lexicographically smallest pair $(\delta(s, v), h)$ — a cheapest route, and among the cheapest, one with the fewest edges. (Keying the queue by d alone is not enough: with zero-weight edges, a vertex could be extracted at the right distance but with too many edges, before a tie with fewer edges is discovered.)

13.13 Run the $|V| - 1$ passes as usual. Then run one extra pass and collect into a set N every vertex v for which some edge (u, v) still relaxes. Do a breadth-first or depth-first search *forward* from every vertex of N and mark everything it reaches: all marked vertices get $-\infty$. Of the unmarked, those with $d[v] = \infty$ get $+\infty$ and the rest keep their finite $d[v]$. Running time $\Theta(VE)$ for the passes plus $\Theta(V + E)$ for the spreading search, so $\Theta(VE)$ overall. **Listing 13.8** implements exactly this.

Listing 13.8 Solution to Exercise 13.13: Bellman–Ford extended so that each vertex is labeled $+\infty$, a finite distance, or $-\infty$, according to whether it is unreachable, safely reachable, or reachable from a negative cycle.

```
#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 u, v; long long w; };

int main() {
    // s -> a -> b -> c -> a is a cycle of weight 2 + (-5) + 1 = -2, and c -> d
    // leaves it, so a, b, c and d all have delta = -infinity.  e is reachable
    // from s without touching the cycle, and f is not reachable at all.
    vector<string> name = {"s", "a", "b", "c", "d", "e", "f"};
    int n = (int)name.size();
    vector<Edge> edges = {
        {0,1,1}, {1,2,2}, {2,3,-5}, {3,1,1}, {3,4,3}, {0,5,4}, {6,5,1}
    };

    vector<long long> d(n, INF);
    d[0] = 0;
    for (int i = 1; i <= n - 1; ++i)
        for (const Edge& e : edges)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) d[e.v] = d[e.u] + e.w;

    // One more pass marks every endpoint that still improves; then we spread
    // that mark forward, because everything reachable from such a vertex is
    // also driven to -infinity.
    vector<bool> minusInf(n, false);
    queue<int> q;
    for (const Edge& e : edges)
        if (d[e.u] < INF && d[e.u] + e.w < d[e.v] && !minusInf[e.v]) {
            minusInf[e.v] = true; q.push(e.v);
        }
    vector<vector<int>> adj(n);
    for (const Edge& e : edges) adj[e.u].push_back(e.v);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adj[u]) if (!minusInf[v]) { minusInf[v] = true; q.push(v); }
    }
```

Listing 13.8 (Continued) *Solution to Exercise 13.13: Bellman–Ford extended so that each vertex is labeled +\infty, a finite distance, or -\infty, according to whether it is unreachable, safely reachable, or reachable from a negative cycle.*

```
    cout << "vertex |  delta(s,v)\n";
    cout << "-------+------------\n";
    for (int v = 0; v < n; ++v) {
        cout << "   " << name[v] << "   |  ";
        if (minusInf[v])      cout << "-infinity\n";
        else if (d[v] >= INF) cout << "+infinity (unreachable)\n";
        else                  cout << d[v] << "\n";
    }
    return 0;
}
```

```
vertex |  delta(s,v)
-------+------------
   s   |  0
   a   |  -infinity
   b   |  -infinity
   c   |  -infinity
   d   |  -infinity
   e   |  4
   f   |  +infinity (unreachable)
```

The graph is that of **Figure 13.2** plus a vertex e reachable from s but not from the cycle, and a vertex f not reachable at all. All three labels appear.

13.14 Add a new vertex q and an edge (q, v) of weight 0 for every $v \in V$. No new cycle is created, because q has no incoming edges, and every original vertex is now reachable from q. Run Bellman–Ford from q on the enlarged graph, which has $|V| + 1$ vertices and $|E| + |V|$ edges; it returns `false` exactly when the original graph contains a negative cycle anywhere. Cost $\Theta((V + 1)(E + V)) = \Theta(VE)$ for a connected graph.

13.15 When the detector finds an edge (u, v) that still relaxes, set $x \leftarrow v$ and follow π backward $|V|$ times: $x \leftarrow \pi[x]$, repeated $|V|$ times. *Why this lands on the cycle:* the predecessor chain from v must enter the cycle after at most $|V| - 1$ steps, since there are only $|V|$ distinct vertices and the chain cannot leave a cycle once inside it; so after $|V|$ steps the current vertex is certainly on the cycle. Then record $y \leftarrow x$ and walk π from x, collecting vertices, until returning to y; reverse the collected list to get the cycle in forward order. Cost $\Theta(V)$ on top of Bellman–Ford.

13.16 Suppose $\delta(s, u) = \delta(s, v)$ and $(u, v) \in E$. The edge form of the triangle inequality gives $\delta(s, v) \leq \delta(s, u) + w(u, v)$, so $\delta(s, u) \leq \delta(s, u) + w(u, v)$, hence $w(u, v) \geq 0$. (The hypothesis of non-negative weights is not even needed for this direction.) The converse fails: in the Dijkstra example graph, $w(x, y) = 2 \geq 0$ but $\delta(s, x) = 5 \neq 7 = \delta(s, y)$. Equality of distances is a much stronger condition than non-negativity of the connecting edge.

References

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

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, 1st ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1993.

[3] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[4] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[5] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[6] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[7] L. R. Ford, Jr., "Network flow theory," RAND Corporation, Santa Monica, CA, USA, Paper P-923, Aug. 1956.

[8] E. F. Moore, "The shortest path through a maze," in *Proc. International Symposium on the Theory of Switching*, Cambridge, MA, USA, 1959, pp. 285–292.

[9] Profitina, "Profitina — AI Visibility (GEO) platform," product documentation, app.profitina.com, 2026.

Chapter 14
Amortized Analysis

By the end of this chapter you will be able to look at a data structure whose operations are sometimes cheap and occasionally catastrophic, and prove a bound on the *whole sequence* that is far better than the bound you get by adding up worst cases one operation at a time. You will be able to explain, to someone who has never heard the word, what *amortized* means and — just as important — what it does not mean: it is not average-case analysis, no probability appears anywhere in it, and the bound it proves holds for *every* sequence of operations, including the one an enemy chose. You will be able to apply all three standard methods — the aggregate method, the accounting method and the potential method — to the same structure and get the same answer three ways. You will be able to take three classic exercises — known by their numbers in the third edition of Cormen et al. as 17.1-3, 17.2-2 and 17.3-6, and found with the same wording in the amortized-analysis chapter of the fourth edition [1] — and work each one through from the first line to the last. You will be able to implement a dynamic table that doubles when it fills and halves when it empties, say exactly why halving at half-full is a disaster and halving at quarter-full is not, and write out the sequence of operations that proves it. And you will be able to answer the question Chapter 11 left hanging: why a disjoint-set forest with union by rank and path compression costs $O(\alpha(n))$ per operation, what α actually is, and why it is at most 4 for every problem that will ever run on a physical machine.

14.1 Where We Left Off

14.1.1 A debt from Chapter 11

Chapter 11 built a disjoint-set forest to run Kruskal's algorithm. It added two tricks — union by rank and path compression — and then said something that must have felt like a dodge:

With both tricks, the cost of a sequence of m operations on n elements is $O(m \cdot \alpha(n))$, where α is the inverse Ackermann function… you cannot bound a single `FindSet` by $\alpha(n)$, because one find genuinely can cost $\Theta(\lg n)$; you can only bound the *average over the whole sequence*.

That sentence contains the whole idea of this chapter in one clause: *you can only bound the average over the whole sequence*. Chapter 11 also promised that Chapter 14's potential method is the tool for proving such things. This chapter honors that promise. Section 14.8 returns to union–find and explains, at a beginner's pace, where the α comes from.

The same chapter made a second promise. Its table of priority-queue costs listed a Fibonacci heap [2] with `DecreaseKey` at "$O(1)$ amortized", and remarked that this is "precisely the kind of claim Chapter 14 exists to prove". By the end of this chapter you will understand exactly what that row of the table is asserting.

14.1.2 What you need to have with you

Very little. You need to know what a stack is (a pile of items where you add and remove only at the top), what an array is, and the notations O, Θ and Ω from Chapter 3. You need to be

comfortable with a summation sign $\sum$, and you need to remember one fact about geometric series, which we will re-derive when we need it. Everything else is built here.

One symbol will appear constantly, so let us fix it now. Throughout this chapter, n is the **number of operations in the sequence we are analyzing** — not the number of elements in the structure. This is a change of habit. In Chapter 5, n was the size of the array being sorted. Here it counts *operations*: pushes, pops, increments, insertions. When we also need the number of elements we will give it a different name, such as s for the number of items on a stack or num for the number of items in a table.

14.2 The Question Amortized Analysis Answers

14.2.1 A bound that is true, and useless

Here is a small data structure. It is a stack, so it has the two usual operations:

- `Push(S, x)` — put the item x on top of the stack S;
- `Pop(S)` — take the top item off the stack S and return it.

Both are plainly constant-time: each touches exactly one slot of the underlying array. Now add a third operation:

- `Multipop(S, k)` — remove the top k items from S, or all of them if there are fewer than k.

`Multipop` is not constant-time. If the stack holds s items and you call `Multipop(S, k)`, the operation removes $\min(s, k)$ items, and removing an item costs one unit, so the cost is $\min(s, k)$.

Now suppose somebody hands you a program that performs n of these operations, in some order you do not control, starting from an empty stack, and asks: how long does the program take?

Here is the analysis that everyone does first. The most expensive a single operation can be is the most expensive a `Multipop` can be. A `Multipop` costs $\min(s, k)$, and the stack can never hold more than n items (because each item had to be pushed, and there are at most n operations). So a single operation costs at most n. There are n operations. Therefore the total is at most

$$n \times n = n^2,$$

and the program runs in $O(n^2)$ time.

Every word of that is true. There is no error in it. And it is almost useless, because the true answer is $O(n)$ — a factor of n better. For $n = 1{,}000{,}000$ the honest-but-crude bound says "up to a trillion steps", and the truth is "under two million". That is the difference between a program you would not dare ship and a program that finishes before you have taken your hand off the mouse.

> **WHERE THE CRUDE BOUND GOES WRONG.** It multiplies the worst cost of *one* operation by the number of operations, as if every operation could be the worst one *at the same time*. It cannot. A `Multipop` that removes a million items can only happen after a million `Push`es have happened — and those pushes were part of the same budget of n operations. The expensive operations and the cheap ones are not independent: **the expensive ones are paid for by the cheap ones that came before**. The crude bound throws that relationship away, and the whole of this chapter is about not throwing it away.

14.2.2 The factor of n, made concrete

Let us make the gap visible with two explicit sequences of exactly n operations.

Sequence one. Push $n-1$ items, then call `Multipop(S, n)` once.

- The $n-1$ pushes cost 1 each, for a total of $n-1$.
- The single `Multipop` removes $n-1$ items, so it costs $n-1$.
- Grand total: $(n-1)+(n-1)=2n-2$.

That single `Multipop` really did cost $\Theta(n)$. The crude bound is not lying about *it*. But the sequence as a whole cost about $2n$, not n^2, because there was only *one* expensive operation and the rest were as cheap as possible.

Sequence two. Try to have many expensive operations. Push $n/2$ items, then perform $n/2$ calls to `Multipop(S, n)`.

- The $n/2$ pushes cost $n/2$ in total.
- The *first* `Multipop` empties the stack, costing $n/2$.
- The remaining $n/2-1$ `Multipop`s find an empty stack and cost 0 each.
- Grand total: $n/2+n/2=n$.

The adversary wanted many expensive `Multipop`s and could only get one. **That is the point.** There is no sequence of n operations on this structure costing more than about $2n$, and Section 14.3.2 proves it. The $O(n^2)$ bound is off by a factor of n — not by a constant, not by a logarithm, by the whole of n.

> **THE QUESTION OF THIS CHAPTER.** *Given a sequence of n operations on a data structure, what is the worst-case total cost of the whole sequence, and hence the worst-case cost per operation when the sequence is taken as a unit?* That last number — total cost divided by n — is what we will call the **amortized cost per operation**. The word "amortize" comes from finance: to amortize a debt is to spread a large one-off payment evenly over many small installments. Here we spread the cost of the rare expensive operation over the many cheap ones that made it possible.

14.2.3 Three costs, and the only inequality that matters

Figure 14.1 lays out the vocabulary with no algebra at all, and it is worth reading slowly because the whole subject is in that one picture.

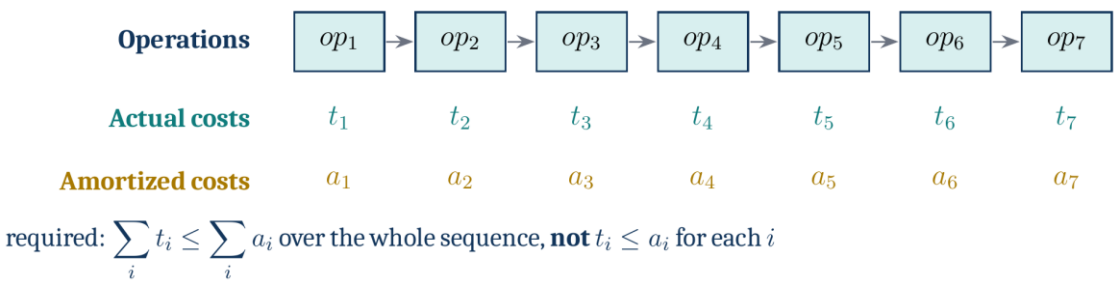


Figure 14.1 Operations $op_1, \ldots, op_7$ with their **actual costs** t_i and the **amortized costs** a_i we assign to them. The requirement is $\sum t_i \le \sum a_i$ over the whole sequence, and emphatically **not** $t_i \le a_i$ for each i.

Read the three rows in order.

The top row is the sequence of operations themselves: $op_1, op_2, \ldots$ — the actual calls the program makes, in the order it makes them.

The middle row is the **actual cost** of each operation. The figure writes it t_i; Cormen et al. [1] and most other books write it c_i, and so will we from Section 14.3 onward, but they mean the same thing: the real, honest number of elementary steps that operation i performs when the program is run. The actual cost of a `Push` is 1. The actual cost of a `Multipop` that removes 37 items is 37.

The bottom row is the **amortized cost**, written a_i. This is a number *we invent*. It is not measured; it is assigned by us, the analysts, and we are free to assign whatever we like — as long as we obey one rule.

The rule is the first requirement shown in the figure:

$$\sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i \quad \text{over the entire sequence, starting from } op_1. \tag{14.1}$$

In words, Equation (14.1) says: **the made-up costs, added up, must come to at least as much as the real costs, added up.** If we can invent a set of amortized costs that satisfies this, and if every one of our invented costs is small, then we have proved a bound on the real total: the real total is at most the invented total, and the invented total is small.

The second statement in the figure is printed in red, because it is the thing beginners get wrong:

Not required: $t_i \leq a_i$ for each i.

We do **not** need each individual amortized cost to cover its own operation's actual cost. A single operation is allowed to blow straight through its amortized allowance — as the `Multipop` that removes a million items certainly does, when its amortized cost is 0. What must hold is only the *running total*, and only from the beginning of the sequence. Some operations save up; later ones spend the savings.

> **DEFINITION.** Let a sequence of n operations be performed on a data structure, and let c_i (written t_i in **Figure 14.1**) be the **actual cost** of the i-th operation. An assignment of numbers $a_1, a_2, \ldots, a_n$ is a valid set of **amortized costs** if

$$\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} a_i \qquad \text{for every } k = 1,2,\ldots,n. \tag{14.2}$$

> When every a_i is bounded by the same value $\hat{c}$, we say the structure supports the operation at **amortized cost** $\hat{c}$, and the total cost of any sequence of n operations is at most $n\hat{c}$.

Notice that Equation (14.2) asks for the inequality at **every prefix** k, not just at the end. This is slightly stronger than the single inequality Equation (14.1) at $k = n$, and it is what the accounting and potential methods actually deliver. It matters: it means you may stop the program at any moment and the bound still holds, which is exactly what you want if the program might be interrupted, or if you want the bound for every k at once.

14.2.4 Amortized analysis is not average-case analysis

Say this out loud once before reading on: **amortized analysis is not average-case analysis.**

They sound alike. Both produce a number that looks like "the cost per operation on average". They are completely different in kind, and confusing them will cost you the correctness of your argument — Chapter 16 works several problems where exactly this distinction decides the answer.

Average-case analysis assumes a *probability distribution* over inputs. It says: if the input is drawn at random according to this distribution, then the *expected* running time is such and such. Its answer is an expectation. Randomized quicksort from Chapter 5 is the standard example: its $O(n\lg n)$ is an expectation over the random pivot choices, and there exist runs — unlikely ones, but real ones — that take $\Theta(n^2)$.

Amortized analysis assumes **nothing**. There is no probability distribution. There is no randomness. There is no "typical input". The bound it proves is a **worst-case** bound on the total cost of the sequence, and it holds for *every* sequence of n operations — including the one that an adversary who has read your source code chose specifically to destroy you.

> **THE MISCONCEPTION THIS CHAPTER EXISTS TO KILL.** "Amortized $O(1)$" does *not* mean "$O(1)$ most of the time, and if you are unlucky it is slow." It means: **there is no sequence of n operations that costs more than $O(n)$ in total.** You may be as unlucky as you like. You may choose the sequence yourself, knowing every line of the implementation. You still cannot make the total exceed the bound. The only thing you cannot conclude is that any *particular* operation is fast — and that is a genuine limitation, which Section 14.4 discusses honestly.

Section 14.4 says this again, after you have seen the aggregate method work, because it lands differently the second time.

14.2.5 The three methods

There are three standard ways to carry out an amortized analysis [1], [3], summarized in **Figure 14.2**.

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.

In one sentence each:

1. **The aggregate method.** Work out the total cost $T(n)$ of any sequence of n operations directly, by a counting argument. Then every operation is assigned the same amortized cost, $T(n)/n$. It is the simplest method and the one to try first, but it gives every operation the same amortized cost, which is sometimes too blunt.
2. **The accounting method.** Assign each *type* of operation its own charge. Charge more than an operation costs, and the surplus is stored as **credit** on some specific object in the structure. When an expensive operation comes along, it pays with credit that was deposited earlier. The obligation is to prove that the credit balance never goes negative.
3. **The potential method.** The same idea as the accounting method, but the prepayment is not attached to individual objects; it is a single number Φ — the **potential** — computed from the

state of the whole structure. The amortized cost is defined as the actual cost plus the change in potential. It is the most flexible of the three, the one that scales to hard structures such as Fibonacci heaps [2] and splay trees [4], and the one Chapter 11 pointed at.

All three give the same answers on the examples in this chapter. They differ in how much work they are, and in how much they tell you.

14.3 The Aggregate Method

14.3.1 What the method is

The aggregate method is the least clever of the three, which is a compliment. It consists of exactly two steps.

Step 1. Show that any sequence of n operations takes $T(n)$ time in the worst case, for some function T that you compute directly.

Step 2. Declare the amortized cost of each operation to be $T(n)/n$.

That is all. Notice what it does *not* do: it does not distinguish between the operations. If a structure supports three different operations, the aggregate method gives all three the same amortized cost, $T(n)/n$, even if one of them is obviously cheaper than the others. That bluntness is the method's only real weakness.

The work is all in Step 1, and the whole art is in finding a counting argument that captures the dependency between operations — the thing the crude bound threw away.

14.3.2 The stack with `Multipop`

Figure 14.3 gives the three operations in pseudocode.

$O(1)$ each

```
PUSH(S, x)
  top[S] ← top[S] + 1
  S[top[S]] ← x
```

```
POP(S)
  if STACK-EMPTY(S)
      then error "underflow"
      else top[S] ← top[S] − 1
      return S[top[S] + 1]
```

cost = number of loop iterations = $\min(s, k)$

```
MULTIPOP(S, k)
  while not STACK-EMPTY(S) and k ≠ 0
      do POP(S)
          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.

Read the three procedures line by line, because we will count their costs exactly.

`Push(S, x)` does two things: it increases the index `top[S]` by one, and it writes x into the slot that index now names. Two steps, independent of how big the stack is. Cost: $O(1)$. We will call it exactly 1 unit, which only changes the answer by a constant factor.

`Pop(S)` checks whether the stack is empty; if it is, it reports an underflow error; if not, it decreases `top[S]` by one and returns the value that used to be on top. Again a fixed number of steps. Cost: 1 unit.

`Multipop(S, k)` loops while the stack is non-empty *and* $k \neq 0$, calling `Pop(S)` and decrementing k each time round. So the number of times round the loop is whichever runs out first — the items or the budget k. If the stack currently holds s items, the loop runs

$$\min(s, k) \text{ times,}$$

and since each iteration is one `Pop`, the actual cost of the call is exactly $\min(s, k)$ units.

Stated plainly: if the stack contains s elements then `Multipop(k)` has time complexity $O(\min(s, k))$, **not** $O(1)$.

The naive bound, once more

We did this in Section 14.2.1, but let us write it out once more in full, so the comparison is exact.

The stack starts empty. We perform n operations, each one of `Push`, `Pop` or `Multipop`, in any order. How large can s get? Every item on the stack was put there by a `Push`, and there are at most n operations, so at most n pushes, so $s \leq n$ always. Hence a single `Multipop` costs at most $\min(n, k) \leq n$. Hence one operation costs $O(n)$. Hence n operations cost $O(n^2)$.

This is the bound obtained from the worst case "for a single Multipop operation", and it gives $O(n^2)$ for the sequence. However, using the aggregate method, we can derive a much tighter upper bound.

The observation that fixes it

Here is the key observation:

The total number of `Pop` operations (including those performed within `Multipop`) cannot exceed the total number of `Push` operations — that is, at most $O(n)$.

Let us prove that carefully, because it is the heart of the matter and it is the kind of argument you will need to reproduce.

Claim. In any sequence of operations starting from an empty stack, the number of times an item is removed from the stack is at most the number of times an item is put on it.

Proof. Look at any single item — call it x. How did x get onto the stack? There is exactly one way: some `Push` put it there. How can x leave the stack? There are exactly two ways: a `Pop`, or one iteration of the loop inside a `Multipop` — and that loop body *is* a `Pop`. So each removal of x is one `Pop`. Can x be removed twice? No: once removed, x is no longer on the stack, and nothing ever puts it back except another `Push`, which would be a different push. So:

- each item is pushed exactly once by the push that created it, and
- each item is popped **at most once** after that push — at most, because the sequence may end with x still sitting on the stack.

Therefore the removals are in one-to-one correspondence with a subset of the pushes, and

$$\text{(number of removals)} \;\leq\; \text{(number of pushes)}. \qquad \blacksquare$$

> **"EACH ELEMENT IS POPPED AT MOST ONCE AFTER IT IS PUSHED."** That single sentence is the whole aggregate analysis of the stack. It is worth memorizing in that form, because the same sentence — with different nouns — proves the amortized bound for a dozen other structures. The general shape is: *find a unit of work that can be charged to an object, and show that each object can be charged at most a bounded number of times over the whole sequence.*

The total, computed

Now the arithmetic. Let the sequence contain p pushes and q removals, where a removal means one item leaving the stack (whether through a `Pop` or inside a `Multipop`). Each of the n operations is a `Push`, a `Pop` or a `Multipop`, so $p \leq n$. The claim gives $q \leq p \leq n$.

The total actual cost is

$$T(n) \;=\; \underbrace{p \times 1}_{\text{the pushes}} \;+\; \underbrace{q \times 1}_{\text{the removals}} \;\leq\; n + n \;=\; 2n.$$

(The bookkeeping inside `Multipop` — testing the loop condition, decrementing k — adds a constant per iteration plus one for the last test, which changes $2n$ into cn for some small constant c. We are working in O-notation, so this does not matter; but be aware that "2" is the clean idealized figure, not a literal instruction count.)

So $T(n) = O(n)$ for any sequence of n operations, and by Step 2 of the aggregate method the amortized cost per operation is

$$\frac{T(n)}{n} \leq \frac{2n}{n} = 2 = O(1).$$

Every one of the three operations — `Push`, `Pop` and `Multipop` — has 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.

Measuring it

Claims about running time should be checked against a running program. **Listing 14.1** is a stack with all three operations, instrumented so that every item leaving the stack is counted, and run on three different 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`.

```
// 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;
}
```

Running it prints:

```
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.
```

Look at sequence A. It pushes as much as it can and then empties the stack in one gigantic `Multipop`; that single operation removes nearly 50,000 items. The total is 199,996 against a ceiling of 200,000 — the adversary has pushed the structure to within four units of the proved bound and cannot go further. Sequence C, an ordinary-looking mix, comes in at 1.29 per operation. In no case does the measurement reach 2, and in no case is it anywhere near $n =$ 100,000, which is what the $O(n^2)$ bound would have allowed.

14.3.3 The k-bit binary counter

The second example is the one to keep in your head, because it recurs throughout computer science: a binary counter that you increment from zero.

The structure is an array $A[0 \ldots k-1]$ of k bits, where `length[A]` $=k$. The bit $A[0]$ is the **least significant** — the ones place — and $A[k-1]$ is the most significant. The number the counter currently represents is

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i.$$

So if $k = 8$ and the bits, from $A[7]$ down to $A[0]$, read 0 0 0 0 1 0 1 1, the value is $8 + 2 + 1 = 11$.

To add one to the counter we do what you do on paper in base ten when you add 1 to 1999: turn the trailing nines into zeros and bump the first non-nine. In base two the "nines" are ones. The procedure `Increment` is **Figure 14.4**.

```
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 it once by hand on $k = 4$, starting from $A = 0\,1\,1\,1$ (the value 7, written most-significant first).

- $i = 0$. Is $i < 4$ and $A[0] = 1$? Yes. So $A[0] \leftarrow 0$ and $i \leftarrow 1$. Array: 0 1 1 0.
- $i = 1$. Is $A[1] = 1$? Yes. So $A[1] \leftarrow 0$, $i \leftarrow 2$. Array: 0 1 0 0.
- $i = 2$. Is $A[2] = 1$? Yes. So $A[2] \leftarrow 0$, $i \leftarrow 3$. Array: 0 0 0 0.

- $i = 3$. Is $A[3] = 1$? No, it is 0. Leave the loop.
- Line 5: is $3 < 4$? Yes. Line 6: $A[3] \leftarrow 1$. Array: 1 0 0 0.

The value went from 7 to 8, and four bits were written. In general, **the cost of one `Increment` is the number of bits it changes**: one write per trailing 1 that gets reset, plus one write for the 0 that gets set.

The 8-bit table

Figure 14.5 shows an 8-bit counter driven from 0 up to 16, with the bit pattern at every step and a running total of the cost.

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 $2 \times 16 = 32$.

Walk down the shaded column with your finger and two things jump out.

First, $A[0]$ is shaded in *every* row. Of course: adding 1 always changes the ones bit. So $A[0]$ flips n times in n increments.

Second, the wide rows — the ones with many shaded cells — are rare, and they get rarer as they get wider. Going from 7 to 8 shades four cells. Going from 15 to 16 shades five. But those are the only rows that wide in the whole table. Between them sit seven rows, and four of those shade a single cell.

That is the dependency the crude bound misses, in picture form. The crude bound says: an `Increment` can flip up to k bits, there are n of them, so the total is $O(nk)$. The correction is a single observation: *not all bits change every time the `Increment` procedure is called!*

How often does bit i flip?

This is the counting argument. Take it slowly.

Bit $A[0]$ changes on **every** increment. In n increments it flips n times.

Bit $A[1]$ changes only when $A[0]$ was 1 and had to be reset, which happens on every *second* increment: going $1 \to 2$, $3 \to 4$, $5 \to 6$, and so on. In n increments it flips $\lfloor n/2 \rfloor$ times.

Bit $A[2]$ changes only when both $A[0]$ and $A[1]$ were 1 — that is, on every *fourth* increment: $3 \to 4$, $7 \to 8$, $11 \to 12$. In n increments it flips $\lfloor n/4 \rfloor$ times.

The pattern is clear, and it can be stated as: *In general, for* $i = 0,1, \ldots, \lfloor \log_2 n \rfloor$, *the bit* $A[i]$ *will change* $\lfloor n/2^i \rfloor$ *times over the course of* n *operations.* And for $i > \lfloor \log_2 n \rfloor$ the bit never changes at all, because the counter never gets that large.

Let us see why $\lfloor n/2^i \rfloor$ is right. Bit $A[i]$ changes exactly when the increment carries all the way up to position i, which happens exactly when the value being incremented has its low i bits all equal to 1 — that is, when the value is congruent to $2^i - 1$ modulo 2^i. Among the n values $0,1, \ldots, n-1$ that we increment, the number of such values is $\lfloor n/2^i \rfloor$. (One in every block of 2^i consecutive integers has this form, and there are $\lfloor n/2^i \rfloor$ complete blocks contained in the first n integers, with the partial block at the end contributing nothing extra to the floor.)

Adding it up

The total number of bit flips over n increments is the sum over all bits of how often each one flips:

$$T(n) = \sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor. \tag{14.3}$$

(We write lg for $\log_2$, as Chapter 3 established.)

Figure 14.6 bounds the sum Equation (14.3) in two lines.

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

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.

Here is that display, one step per line, with the reason for each step.

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor \le \sum_{i=0}^{\lfloor \lg n \rfloor} \frac{n}{2^i}$$

Why: $\lfloor x \rfloor \le x$ for every real x. Dropping the floor can only increase each term.

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \frac{n}{2^i} < \sum_{i=0}^{\infty} \frac{n}{2^i}$$

Why: every term is positive, so extending the sum to infinity adds more positive terms and strictly increases the value.

$$\sum_{i=0}^{\infty} \frac{n}{2^i} = n \sum_{i=0}^{\infty} \frac{1}{2^i}$$

Why: n does not depend on i, so it can be pulled out in front of the sum.

$$n \sum_{i=0}^{\infty} \frac{1}{2^i} \;=\; n \cdot 2 \;=\; 2n.$$

Why: this is the infinite geometric series $1 + 1/2 + 1/4 + 1/8 + \cdots$. If you have not met it, here is the one-line proof. Let $S = 1 + 1/2 + 1/4 + \cdots$. Then $1/2\,S = 1/2 + 1/4 + 1/8 + \cdots$, which is S with its first term removed, so $1/2\,S = S - 1$. Solving, $1/2\,S = 1$, hence $S = 2$.

So

$$T(n) \;<\; 2n,$$

and the aggregate method gives an amortized cost per `Increment` of

$$\frac{T(n)}{n} \;<\; \frac{2n}{n} \;=\; 2 \;=\; O(1).$$

> **IN PLAIN LANGUAGE.** An increment can flip as many as k bits, so a single increment can be slow. But to flip k bits you must first have set those k bits, one at a time, over the previous $2^k - 1$ increments. The expensive increments are exactly the ones that a long stretch of cheap increments has been paying for. Averaged over the sequence, fewer than two bits flip per increment — no matter how large k is.

Notice one more thing: the bound $2n$ does not mention k at all. A 64-bit counter and an 8-bit counter cost the same per increment on average. This is why incrementing a loop counter is treated as a constant-time operation in the RAM model of Chapter 2, even though a carry can in principle ripple through the whole register.

Measuring it

Listing 14.2 implements `Increment` exactly as **Figure 14.4** writes it, attaches a flip counter to every bit, and first reproduces the 8-bit table of **Figure 14.5** — the same "Total Cost" column, computed rather than copied.

Listing 14.2 The k-bit binary counter with a per-bit flip counter. Compile with `g++ -std=c++17 -O2 -o counter counter.cpp`.

```
// A k-bit binary counter, implemented exactly as the INCREMENT
// pseudocode, with a flip counter attached.  The program first reproduces
// the 8-bit table, then measures the amortized cost over a long
// run and checks it against the bound proved in the text.
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

class BitCounter {
public:
    explicit BitCounter(int k) : A(k, 0), flips(k, 0) {}

    // INCREMENT(A): returns the actual cost -- the number of bits flipped.
    long long increment() {
        long long cost = 0;
        size_t i = 0;
        while (i < A.size() && A[i] == 1) {   // reset every trailing 1
            A[i] = 0; ++flips[i]; ++cost; ++i;
        }
        if (i < A.size()) {                   // set the first 0 we reached
            A[i] = 1; ++flips[i]; ++cost;
        }
        total += cost; ++nOps;
        return cost;
    }

    unsigned long long value() const {        // only valid while k <= 63
        unsigned long long v = 0;
        for (size_t i = A.size(); i-- > 0; ) v = 2 * v + A[i];
        return v;
    }
    int bit(int i) const { return A[i]; }
    long long flipsOf(int i) const { return flips[i]; }
    long long actualCost() const { return total; }
    long long ops() const { return nOps; }
    size_t width() const { return A.size(); }

private:
    vector<int> A;            // the bit array A[0..k-1]; A[0] is least significant
    vector<long long> flips;
    long long total = 0, nOps = 0;
};
```

Listing 14.2 (Continued) *The k-bit binary counter with a per-bit flip counter. Compile with g++ -std=c++17 -O2 -o counter counter.cpp.*

```
int main() {
    // ---- Part 1: the 8-bit table, columns A[7]..A[0] and Total Cost.
    cout << "Counter";
    for (int i = 7; i >= 0; --i) cout << "  A[" << i << "]";
    cout << "   Total Cost\n";
    BitCounter C(8);
    cout << setw(5) << 0 << "  ";
    for (int i = 7; i >= 0; --i) cout << setw(5) << C.bit(i) << " ";
    cout << setw(9) << 0 << "\n";
    for (int v = 1; v <= 16; ++v) {
        C.increment();
        cout << setw(5) << v << "  ";
        for (int i = 7; i >= 0; --i) cout << setw(5) << C.bit(i) << " ";
        cout << setw(9) << C.actualCost() << "\n";
    }

    // ---- Part 2: how often each bit really flips, against floor(n/2^i).
    const long long n = 1000000;
    BitCounter D(32);
    for (long long t = 0; t < n; ++t) D.increment();
    cout << "\nAfter n = " << n << " increments on a 32-bit counter:\n";
    cout << " bit i   measured flips   predicted floor(n/2^i)\n";
    for (int i = 0; i < 24; ++i)
        cout << setw(6) << i << setw(17) << D.flipsOf(i)
             << setw(25) << (n >> i) << "\n";
    cout << "\ntotal actual cost = " << D.actualCost()
         << "   ceiling 2n = " << 2 * n << "\n";
    cout << "measured amortized cost per INCREMENT = " << fixed << setprecision(6)
         << (double)D.actualCost() / (double)D.ops() << "\n";
    return 0;
}
```

The first half of the output is the table of **Figure 14.5**, recomputed:

```
Counter  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
```

Compare that Total Cost column with **Figure 14.5**: 0,1,3,4,7,8,10,11,15,16,18,19,22,23,25,26,31. Identical. The hand-drawn table is exactly right, which is a pleasant thing to be able to confirm.

The second half checks the flip formula against reality on a million increments:

```
After n = 1000000 increments on a 32-bit counter:
 bit i   measured flips   predicted floor(n/2^i)
     0          1000000                  1000000
     1           500000                   500000
     2           250000                   250000
     3           125000                   125000
     4            62500                    62500
     5            31250                    31250
     6            15625                    15625
     7             7812                     7812
     8             3906                     3906
     9             1953                     1953
    10              976                      976
    11              488                      488
    12              244                      244
    13              122                      122
    14               61                       61
    15               30                       30
    16               15                       15
    17                7                        7
    18                3                        3
    19                1                        1
    20                0                        0
    21                0                        0
    22                0                        0
    23                0                        0

total actual cost = 1999993   ceiling 2n = 2000000
measured amortized cost per INCREMENT = 1.999993
```

Every measured flip count equals $\lfloor n/2^i \rfloor$ exactly — including the rows from $i = 20$ on, where the counter simply never reaches that high and the bit never moves. The total, 1,999,993, is seven units below the ceiling $2n = 2{,}000{,}000$, so the bound $T(n) < 2n$ is both correct and tight to within a rounding error. Notice also the *crude* bound for comparison: $O(nk) = 10^6 \times 32 = 3.2 \times 10^7$, sixteen times larger than the truth.

14.3.4 Exercise 17.1-3, worked in full

This exercise is the standard drill for the aggregate method, and it is worth every minute.

> **EXERCISE 17.1-3.** 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.

A small table, **Figure 14.7**, makes the pattern concrete before any algebra starts.

- 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.

Understanding the cost function

No data structure is described, and none is needed: the exercise hands us the cost of every operation directly. Write c_i for the cost of operation i. Then

$$c_i = \begin{cases} i & \text{if } i = 2^j \text{ for some integer } j \geq 0, \\ 1 & \text{otherwise.} \end{cases} \tag{14.4}$$

In Equation (14.4), $c_1 = 1$ (and $1 = 2^0$, so this is a power of 2 and it charges $i = 1$, which happens to be 1 anyway), $c_2 = 2$, $c_3 = 1$, $c_4 = 4$, $c_5 = c_6 = c_7 = 1$, $c_8 = 8$, and so on. This is a caricature of a real structure — the doubling table of Section 14.7 behaves almost exactly like this — and the caricature is deliberate: it isolates the pattern.

The worst-case cost of a *single* operation is n (when $i = n$ and n is a power of 2), so the crude bound says $O(n^2)$ again. Let us do better.

Splitting the sum

The total cost of n operations is $\sum_{i=1}^{n} c_i$. The trick — and it is the standard trick — is to split the operations into the expensive ones and the cheap ones, and count each group separately.

The cheap part. *Every* operation, expensive or not, is charged at least 1. There are n operations. So a baseline of n is paid by the sequence, no matter what. Write that as $\sum_{i=1}^{n} 1 = n$.

The expensive part. The operations that are powers of two pay *extra*. Operation $i = 2^j$ pays 2^j in total, of which 1 was already counted in the baseline, so the extra is $2^j - 1$.

Which powers of two occur? Those 2^j with $2^j \leq n$. Taking lg of both sides, $j \leq \lg n$, and since j is an integer, $j \leq \lfloor \lg n \rfloor$. So j runs over $0, 1, 2, \ldots, \lfloor \lg n \rfloor$.

Putting the two parts together gives the first line of the solution in **Figure 14.8**:

$$\text{Total} = \sum_{j=0}^{\lfloor \lg n \rfloor} (2^j - 1) + \sum_{i=1}^{n} 1. \tag{14.5}$$

$$\begin{aligned}
\textit{Total} &= \sum_{j=0}^{\lfloor \lg n \rfloor} (2^j - 1) + \sum_{i=1}^{n} 1 && \text{powers of two pay } 2^j \text{ instead of } 1 \\
&= \left(2^{\lfloor \lg n \rfloor + 1} - 1\right) - \left(\lfloor \lg n \rfloor + 1\right) + n && \text{geometric series: } \sum_{j=0}^{m} x^j = \frac{x^{m+1} - 1}{x - 1} \\
&= 2^{\lfloor \lg n \rfloor + 1} + n - \lfloor \lg n \rfloor - 2 \\
&\leq 2n + n - \lfloor \lg n \rfloor - 2 && \text{since } 2^{\lfloor \lg n \rfloor} \leq 2^{\lg n} = n \\
&= 3n - \lfloor \lg n \rfloor - 2.
\end{aligned}$$

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 Total $\leq 3n - \lfloor \lg n \rfloor - 2$, with the geometric-series formula and the logarithm identity it needs.

Evaluating it, one line at a time

The solution uses the closed form for a finite geometric series, which we should prove rather than accept. For $x \neq 1$,

$$\sum_{k=0}^{m} x^k = \frac{x^{m+1} - 1}{x - 1}. \tag{14.6}$$

Proof. Let $S = 1 + x + x^2 + \cdots + x^m$. Multiply by x: $xS = x + x^2 + \cdots + x^{m+1}$. Subtract: $xS - S = x^{m+1} - 1$, because every term except the last of xS and the first of S cancels. So $S(x-1) = x^{m+1} - 1$, and dividing by $x - 1$ gives the formula. ■

With $x = 2$ and $m = \lfloor \lg n \rfloor$, Equation (14.6) gives

$$\sum_{j=0}^{\lfloor \lg n \rfloor} 2^j = \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2 - 1} = 2^{\lfloor \lg n \rfloor + 1} - 1.$$

The -1 inside the bracket of Equation (14.5) is subtracted once for each value of j, and j takes $\lfloor \lg n \rfloor + 1$ values (counting $j = 0$). So

$$\sum_{j=0}^{\lfloor \lg n \rfloor} (2^j - 1) = (2^{\lfloor \lg n \rfloor + 1} - 1) - (\lfloor \lg n \rfloor + 1).$$

And the baseline is $\sum_{i=1}^{n} 1 = n$. So

$$\text{Total} = 2^{\lfloor \lg n \rfloor + 1} - 1 - (\lfloor \lg n \rfloor + 1) + n,$$

which is the second line of **Figure 14.8**. Tidying the two constants, $-1 - 1 = -2$:

$$\text{Total} = 2^{\lfloor \lg n \rfloor + 1} + n - \lfloor \lg n \rfloor - 2. \tag{14.7}$$

That is the third line, and Equation (14.7) is an **exact** formula — not a bound. We will check it numerically in a moment.

Turning it into a bound

The only awkward term is $2^{\lfloor \lg n \rfloor + 1}$. Removing the floor can only increase the exponent:

$$2^{\lfloor \lg n \rfloor + 1} \leq 2^{\lg n + 1}.$$

The general identity $a^{\log_b n} = n^{\log_b a}$, noted in **Figure 14.8**, would handle this. In our case it is simpler than that: $2^{\lg n + 1} = 2^{\lg n} \cdot 2^1 = n \cdot 2 = 2n$, since $2^{\lg n} = n$ by the definition of lg. So

$$\text{Total} \leq 2n + n - \lfloor \lg n \rfloor - 2 = 3n - \lfloor \lg n \rfloor - 2,$$

the last line of the solution. Since $\lfloor \lg n \rfloor \geq 0$ for $n \geq 1$, we may drop it too and conclude

$$\text{Total} \leq 3n - 2 = O(n).$$

By Step 2 of the aggregate method, the amortized cost per operation is

$$\frac{\text{Total}}{n} \leq \frac{3n-2}{n} < 3 = O(1),$$

and the conclusion reads: *the total cost for n operations is $O(n)$; the resulting amortized cost per operation is $O(1)$.*

> **WHY 3 AND NOT SOMETHING WORSE.** Every operation pays 1. The expensive ones pay extra, but the extras are $1,2,4,8,\ldots$ up to at most n, and that series **doubles**, so its sum is dominated by its last term and comes to less than $2n$. One unit each for n operations, plus at most $2n$ for all the extras together, gives $3n$. Whenever you see costs that double, expect the total to be about twice the largest one, and expect a constant amortized cost.

Checking it against a program

An exact formula deserves an exact test. **Listing 14.3** adds the costs up directly and compares the running total with both the closed form and the bound.

Listing 14.3 Exercise 17.1-3 checked numerically against the closed form and the bound. Compile with `g++ -std=c++17 -O2 -o ex1713 ex1713.cpp`.

```
// Exercise 17.1-3.  The i-th operation costs i when i is an exact power
// of 2, and 1 otherwise.  This program adds the costs up directly and
// compares the running total with the closed form derived in the text,
//     Total(n) = 2^(floor(lg n)+1) + n - floor(lg n) - 2,
// and with the bound  Total(n) <= 3n - floor(lg n) - 2.
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

bool isPowerOfTwo(long long i) { return i > 0 && (i & (i - 1)) == 0; }

long long cost(long long i) { return isPowerOfTwo(i) ? i : 1; }

int floorLg(long long n) {                  // floor(log2 n) for n >= 1
    int k = 0;
    while ((1LL << (k + 1)) <= n) ++k;
    return k;
}

long long closedForm(long long n) {
    int k = floorLg(n);
    return (1LL << (k + 1)) + n - k - 2;
}

int main() {
    cout << "   n   sum of costs   closed form   3n-floor(lg n)-2   amortized\n";
    long long running = 0;
    long long checkpoints[] = {1, 2, 3, 4, 5, 8, 16, 100, 1000, 1024, 100000, 1000000};
    int next = 0;
    for (long long i = 1; i <= 1000000; ++i) {
        running += cost(i);
        if (next < 12 && i == checkpoints[next]) {
            int k = floorLg(i);
            cout << setw(8) << i << setw(15) << running
                 << setw(14) << closedForm(i)
                 << setw(19) << 3 * i - k - 2
                 << setw(12) << fixed << setprecision(4)
                 << (double)running / (double)i << "\n";
            ++next;
        }
    }
    cout << "\nThe first two numeric columns agree at every checkpoint, so the\n"
            "closed form is right; the third is never smaller than the first,\n"
            "so the bound holds; the last column stays below 3, so the\n"
            "amortized cost is O(1).\n";
```

Listing 14.3 (Continued) *Exercise 17.1-3 checked numerically against the closed form and the bound. Compile with g++ -std=c++17 -O2 -o ex1713 ex1713.cpp.*

```
    return 0;
}
```

```
    n   sum of costs   closed form   3n-floor(lg n)-2   amortized
        1              1             1                   1      1.0000
        2              3             3                   3      1.5000
        3              4             4                   6      1.3333
        4              8             8                   8      2.0000
        5              9             9                  11      1.8000
        8             19            19                  19      2.3750
       16             42            42                  42      2.6250
      100            220           220                 292      2.2000
     1000           2013          2013                2989      2.0130
     1024           3060          3060                3060      2.9883
   100000         231054        231054              299982      2.3105
  1000000        2048555       2048555             2999979      2.0486

The first two numeric columns agree at every checkpoint, so the
closed form is right; the third is never smaller than the first,
so the bound holds; the last column stays below 3, so the
amortized cost is O(1).
```

Three things to read out of that table. First, the "sum of costs" and "closed form" columns are identical at every checkpoint, so the derivation is right. Second, the bound $3n - \lfloor \lg n \rfloor - 2$ is never below the true total, and the two are *equal* exactly when n is a power of two — look at $n = 1,2,4,8,16,1024$. So the bound is tight, and cannot be improved. Third, the measured amortized cost wanders between about 1.3 and 2.99 but never reaches 3, which is what "$O(1)$ with constant 3" means in practice.

Compare the row $n = 5$ with the table in **Figure 14.7**: it gives a total of 9 for five operations, and the program gives 9. The arithmetic checks out.

14.3.5 What the aggregate method cannot do

Before moving on, be clear about the method's limits.

It gives one number for all operations. On the stack, `Push`, `Pop` and `Multipop` all come out at amortized cost 2, even though a `Push` obviously costs exactly 1 and always will. If you want to say "pushes are cheap and multipops are free", the aggregate method cannot say it for you.

It needs a global counting argument. On the stack and the counter, one existed and was pretty. On a splay tree [4] or a Fibonacci heap [2], no such argument is known, and the aggregate method simply fails. This is why the other two methods exist.

It tells you nothing about *when* the cost is incurred. If your program has a deadline — a real-time system, an interactive interface, a network handler — knowing that the total is $O(n)$ is no comfort if one operation in the middle takes half a second. Amortized bounds are about throughput, not latency. Section 14.11 returns to this.

14.4 Amortized Is Not Average-Case — Said Again

You have now seen two complete amortized analyses. This is the moment to say the thing again, because now you have something to attach it to.

Amortized analysis is not average-case analysis.

Go back and look at the proof in Section 14.3.2 — "each element is popped at most once after it is pushed". Where is the probability? There is none. We did not assume the sequence was random. We did not assume pushes were more likely than pops. We proved a statement about *every*

sequence: whatever order the n operations come in, chosen by whoever you like for whatever reason, the total is at most $2n$.

Now look at the counter argument in Section 14.3.3. Again: where is the probability? Nowhere. We counted exactly how many times each bit flips, for the specific sequence "increment n times from zero". There is only one such sequence. There is nothing to average over.

> **TWO SENTENCES THAT SOUND THE SAME AND ARE NOT.**
>
> *"Randomized quicksort runs in $O(n\lg n)$ expected time."* — Meaning: if you run it many times on the same input, the average of the running times will be $O(n\lg n)$. Some individual runs will be worse. With bad luck, a single run can take $\Theta(n^2)$. The guarantee is probabilistic.
>
> *"`Multipop` runs in $O(1)$ amortized time."* — Meaning: for every sequence of n stack operations, the total time is $O(n)$. There is no luck involved. You cannot construct a sequence that beats the bound, and neither can anyone else. The guarantee is absolute.

Here is a test you can apply. Ask: **could an adversary who knows my entire implementation defeat this bound?** For an average-case bound, often yes — an adversary who knows your pivot rule can feed randomized quicksort an input that makes it slow, if he can also predict your random numbers. For an amortized bound, never. The amortized bound is a worst-case bound on a sum.

There is a second confusion worth heading off, because it is subtler. Amortized analysis is also **not** the same as "usually fast". A structure can have a terrible amortized bound and still be fast in ordinary use, and a structure can have a beautiful amortized bound and still stall at an awkward moment. The amortized bound is a statement about the *total*, and nothing else. What it buys you is the right to write $n \times O(1)$ instead of $n \times O(n)$ when you are costing a loop — which is usually exactly what you wanted, and occasionally is not.

14.5 The Accounting Method

14.5.1 Charges and credit

The aggregate method computes a total and divides. The accounting method works the other way round: it *decides in advance* what each operation will be charged, and then proves that the charges are enough.

The method rests on three rules:

Amortized cost in the accounting method consists of the actual cost and credits, which can be deposited or withdrawn.

To demonstrate that the average cost per operation in the worst case is low, the total amortized cost over a sequence of operations must serve as an upper bound on the total actual cost of that sequence.

The total credit associated with a data structure must always remain non-negative.

Here is the metaphor, which is worth taking literally because the bookkeeping really is bookkeeping.

Imagine that every elementary step of work costs one rupiah to perform. When an operation runs, we hand it an allowance — its **amortized cost** $\hat{c}_i$, which we choose. The operation spends its actual cost c_i on doing the work. If we gave it more than it spent, the surplus $\hat{c}_i - c_i$ does not disappear: we deposit it in the structure, attached to some specific object. If we gave it less than it

spent, it makes up the difference by withdrawing from deposits that earlier operations left behind.

The accumulated deposits are the **credit** in the structure. Write it C_k after k operations:

$$C_k = \sum_{i=1}^{k} \hat{c}_i - \sum_{i=1}^{k} c_i.$$

14.5.2 The invariant, and why it is what you must prove

Everything turns on one requirement.

THE CREDIT INVARIANT. For every $k = 1,2, \dots, n$,

$$C_k = \sum_{i=1}^{k} \hat{c}_i - \sum_{i=1}^{k} c_i \geq 0.$$

(14.8)

In words: **the structure never goes into debt.** At no point may the work done so far exceed the allowances handed out so far.

Why does this give us what we want? Rearrange the inequality:

$$\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \hat{c}_i.$$

That is precisely the requirement Equation (14.2) of Section 14.2.3. So if the credit never goes negative, the chosen amortized costs are legitimate, and the total actual cost of the sequence is bounded by the total of the allowances. If every allowance is at most some constant $\hat{c}$, the total is at most $n\hat{c} = O(n)$.

Why is proving $C_k \geq 0$ enough, and why must it hold at every k and not merely at $k = n$? Because the sequence can stop anywhere. If the credit were allowed to dip below zero at step 500 and recover by step 1000, then the prefix of 500 operations would cost more than we had allowed for it — and a program that performs only those 500 operations is a perfectly legal program. The bound has to hold for every prefix.

How do you actually prove it? Not by computing C_k in closed form — that is usually hopeless. You prove it by finding a **concrete physical interpretation of the credit**: a rule that says where each stored rupiah is sitting, such that the amount stored is visibly non-negative. This is the craft of the method, and both examples below show it.

THE TRAP. It is tempting to say "the charges are obviously big enough" and move on. They are not obviously anything. The accounting method is only a proof if you *locate* the credit — say which object holds it — and then observe that the count of such objects cannot be negative. An assertion that the balance stays positive, with no account of where the money is, proves nothing.

14.5.3 The stack, by the accounting method

Figure 14.9 gives the assignment.

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 $O(n)$.

The assignment is startling the first time you meet it. `Pop` is charged **zero**. `Multipop`, which can remove a million items, is charged **zero**. Only `Push` is charged, and it is charged double what it costs.

Here is why it works, and this is the "concrete physical interpretation" the previous section demanded.

The credit rule: every item on the stack carries exactly 1 rupiah of credit.

Now check the rule survives each operation.

`Push`. Charged 2. It spends 1 actually putting the item on the stack. The remaining 1 is placed *on that item* and travels with it. The rule still holds: the new item has 1 rupiah, and nothing else changed. Credit goes up by exactly 1, and so does the number of items.

`Pop`. Charged 0. It must spend 1 to remove an item. Where does that rupiah come from? From the item being removed — which, by the rule, is carrying exactly 1. The item pays for its own removal and then leaves, taking nothing with it. Credit goes down by exactly 1, and so does the number of items. The rule still holds for the items that remain.

`Multipop(S,k)`. Charged 0. It removes $\min(s, k)$ items and must spend $\min(s, k)$. Each of those items is carrying exactly 1 rupiah by the rule, so each pays for its own removal. Nothing is borrowed. Credit falls by $\min(s, k)$, exactly as the number of items does.

Why the credit can never go negative. After any prefix of operations, the total credit equals the number of items currently on the stack. The number of items on a stack cannot be negative. Therefore the credit cannot be negative. That is the entire proof, and notice how the physical interpretation did all the work: we never had to compute a sum.

> **READ IT IN ONE SENTENCE.** *Each item pays, at the moment it is pushed, for both its arrival and its eventual departure.* Whether it departs by `Pop` or inside a `Multipop` makes no difference — the fare is prepaid. The maximum charge to any operation is 2, so the amortized cost of all three operations is $O(1)$, and n operations cost at most $2n$. This is the same $2n$ the aggregate method found in Section 14.3.2 — two methods, one answer.

14.5.4 The binary counter, by the accounting method

The assignment for the counter is:

- setting a bit to 1 — charge **2**;
- setting a bit to 0 (a reset) — charge **0**.

And the justification in one line: *The number of 1s in the counter is never negative; therefore, the total credit always remains non-negative.*

Let us expand that into the full argument.

The credit rule: every bit that is currently 1 carries exactly 1 rupiah of credit.

Recall what one `Increment` actually does: it resets some number t of trailing 1 bits to 0, and then sets one 0 bit to 1 (unless it overflows the array, in which case it sets nothing). The actual cost is $t + 1$ writes in the normal case.

The single set-to-1. Charged 2. It spends 1 doing the write. The other 1 is deposited on the bit it just set, which is now a 1 bit — exactly as the rule requires.

The t resets. Charged 0 each. Each reset must spend 1 doing the write. Each of those bits was a 1 immediately before, and by the rule was carrying exactly 1 rupiah. That rupiah pays for the reset. The bit becomes 0 and carries nothing, again as the rule requires.

So each `Increment` is charged at most 2, regardless of how many bits it flips. The `Increment` that takes the counter from $2^{20} - 1$ to 2^{20} flips twenty-one bits and is still charged 2: twenty of those flips are resets, paid for by credit sitting on those twenty bits, deposited one at a time over the previous million increments.

Why the credit can never go negative. After any prefix of increments, the total credit equals the number of 1 bits in the counter. A count of 1 bits cannot be negative. Therefore the credit cannot be negative. ■

Total charge for n increments: at most $2n$. Same answer as the aggregate method, third time.

> **THE ONE CASE THE RULE DOES NOT COVER.** If the counter overflows — every bit is 1 and `Increment` runs off the end of the array — then no bit is set to 1, only resets happen, and the operation is charged 0 while spending k. But all k of those resets are paid by the k rupiah sitting on the k one-bits, so the invariant survives; the credit simply drops to zero. The analysis is unharmed. A k-bit counter that has overflowed has, in a real sense, spent everything it saved.

Watching the credit

An invariant you can watch is an invariant you believe. **Listing 14.4** runs both structures and prints the bank balance after every operation, alongside the quantity it is supposed to equal.

Listing 14.4 The accounting method instrumented: the credit balance printed after every operation, next to the quantity the proof claims it equals. Compile with `g++ -std=c++17 -O2 -o credit credit.cpp`.

```
// The accounting method made visible.  Two structures are run side by
// side -- the MULTIPOP stack and the binary counter -- and after every
// single operation the program prints, or at least checks, the bank
// balance:  credit = (amortized charged so far) - (actual cost so far).
// The whole argument of the accounting method is that this number never
// goes below zero.  Here we watch it not go below zero.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;

// ---------- Structure 1: the stack, charged 2 for PUSH and 0 for the rest.
struct AccountedStack {
    vector<int> data;
    long long actual = 0, charged = 0;

    void push(int x)      { data.push_back(x); actual += 1; charged += 2; }
    void pop()            { if (!data.empty()) { data.pop_back(); actual += 1; } charged +=
0; }
    int  multipop(int k)  { int r = 0;
                            while (!data.empty() && k != 0) { data.pop_back(); ++actual; --
k; ++r; }
                            charged += 0; return r; }
    long long credit() const { return charged - actual; }
};

// ---------- Structure 2: the counter, charged 2 for each bit set to 1.
struct AccountedCounter {
    vector<int> A;
    long long actual = 0, charged = 0;
    explicit AccountedCounter(int k) : A(k, 0) {}

    void increment() {
        size_t i = 0;
        while (i < A.size() && A[i] == 1) {   // resetting a 1 costs 1, charged 0
            A[i] = 0; ++actual; ++i;
        }
        if (i < A.size()) {                   // setting a 0 costs 1, charged 2
            A[i] = 1; ++actual; charged += 2;
        }
    }
    long long ones() const { long long c = 0; for (int b : A) c += b; return c; }
    long long credit() const { return charged - actual; }
};
```

Listing 14.4 (Continued) *The accounting method instrumented: the credit balance printed after every operation, next to the quantity the proof claims it equals. Compile with g++ -std=c++17 -O2 -o credit credit.cpp.*

```
int main() {
    cout << "PART 1  Stack: PUSH charged 2, POP and MULTIPOP charged 0.\n";
    cout << "op                     items  actual  charged  credit  (= items?)\n";
    AccountedStack S;
    auto show = [&](const string& what) {
        cout << left << setw(22) << what << right
             << setw(6) << S.data.size() << setw(8) << S.actual
             << setw(9) << S.charged << setw(8) << S.credit()
             << (S.credit() == (long long)S.data.size() ? "   yes" : "   NO") << "\n";
    };
    S.push(1); show("PUSH 1");
    S.push(2); show("PUSH 2");
    S.push(3); show("PUSH 3");
    S.push(4); show("PUSH 4");
    S.pop();   show("POP");
    S.multipop(2); show("MULTIPOP 2");
    S.push(9); show("PUSH 9");
    S.multipop(100); show("MULTIPOP 100");

    long long minCredit = 0;
    unsigned long long seed = 7;
    auto rnd = [&seed]() { seed = seed * 6364136223846793005ULL + 1442695040888963407ULL;
                           return (unsigned)(seed >> 33); };
    AccountedStack T;
    for (int t = 0; t < 200000; ++t) {
        unsigned r = rnd() % 100;
        if (r < 55) T.push((int)r);
        else if (r < 85) T.pop();
        else T.multipop((int)(rnd() % 500));
        if (T.credit() < minCredit) minCredit = T.credit();
    }
    cout << "\n200000 random stack operations: lowest credit ever seen = "
         << minCredit << "; final credit = " << T.credit()
         << ", final stack size = " << T.data.size() << "\n";

    cout << "\nPART 2  Counter: charge 2 for every bit set to 1, 0 for every reset.\n";
    cout << "  n   value  ones  actual  charged  credit  (= ones?)\n";
    AccountedCounter C(16);
    for (int n = 1; n <= 16; ++n) {
        C.increment();
        cout << setw(4) << n << setw(8) << n << setw(6) << C.ones()
             << setw(8) << C.actual << setw(9) << C.charged
             << setw(8) << C.credit()
             << (C.credit() == C.ones() ? "     yes" : "     NO") << "\n";
    }
    AccountedCounter D(32);
```

Listing 14.4 (Continued) *The accounting method instrumented: the credit balance printed after every operation, next to the quantity the proof claims it equals. Compile with g++ -std=c++17 -O2 -o credit credit.cpp.*

```
    long long worst = 0;
    for (long long t = 0; t < 1000000; ++t) {
        D.increment();
        if (D.credit() < worst) worst = D.credit();
    }
    cout << "\n1000000 increments: lowest credit ever seen = " << worst
         << "; final credit = " << D.credit()
         << ", number of 1 bits = " << D.ones() << "\n";
    cout << "actual cost = " << D.actual << ", charged = " << D.charged
         << " (= 2n, the bound proved in the text)\n";
    return 0;
}
```

```
PART 1  Stack: PUSH charged 2, POP and MULTIPOP charged 0.
op                      items  actual  charged  credit  (= items?)
PUSH 1                      1       1        2       1   yes
PUSH 2                      2       2        4       2   yes
PUSH 3                      3       3        6       3   yes
PUSH 4                      4       4        8       4   yes
POP                         3       5        8       3   yes
MULTIPOP 2                  1       7        8       1   yes
PUSH 9                      2       8       10       2   yes
MULTIPOP 100                0      10       10       0   yes

200000 random stack operations: lowest credit ever seen = 0; final credit = 9, final stack
size = 9

PART 2  Counter: charge 2 for every bit set to 1, 0 for every reset.
  n   value  ones  actual  charged  credit  (= ones?)
   1       1     1       1        2       1     yes
   2       2     1       3        4       1     yes
   3       3     2       4        6       2     yes
   4       4     1       7        8       1     yes
   5       5     2       8       10       2     yes
   6       6     2      10       12       2     yes
   7       7     3      11       14       3     yes
   8       8     1      15       16       1     yes
   9       9     2      16       18       2     yes
  10      10     2      18       20       2     yes
  11      11     3      19       22       3     yes
  12      12     2      22       24       2     yes
  13      13     3      23       26       3     yes
  14      14     3      25       28       3     yes
  15      15     4      26       30       4     yes
  16      16     1      31       32       1     yes

1000000 increments: lowest credit ever seen = 0; final credit = 7, number of 1 bits = 7
actual cost = 1999993, charged = 2000000 (= 2n, the bound proved in the text)
```

Read the “credit” column against the “items” column in Part 1, and against the “ones” column in Part 2. They are equal at every single step — which is the credit rule, verified rather than asserted. The lowest credit ever observed, across 200,000 random stack operations and 1,000,000 increments, is 0: the balance touches the floor repeatedly and never goes through it.

And notice the “actual” column in Part 2: 1,3,4,7,8,10,11,15, … ,31. It is the Total Cost column of **Figure 14.5** once more, arrived at from a completely different program.

14.5.5 Exercise 17.2-2, worked in full

The exercise is: **redo Exercise 17.1-3 using the accounting method.** **Figure 14.10** summarizes the solution.

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

$$\mathit{Total}_i = 3i - \big(2(i - j) + i - k - 2\big) = 2j + k + 2 \;\geq\; 2.$$

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.

The charge

The solution opens with the decision: *the amortized cost per operation is 3.* So the rule is simply **charge every operation 3**, whatever it is. Cheap operations that cost 1 deposit 2. Expensive ones that cost i withdraw $i - 3$.

We must now prove the credit never goes negative.

The credit after i operations

By definition the credit is the total charged minus the total actually spent:

$$\text{Total}_i \;=\; \sum_{j=1}^{i} 3 \;-\; \text{ActualCost}_i \;=\; 3i - \text{ActualCost}_i,$$

which is the first displayed line of **Figure 14.10**. And we already know ActualCost_i — it is the exact formula Equation (14.7) from Exercise 17.1-3, with i in place of n:

$$\text{ActualCost}_i \;=\; 2^{\lfloor \lg i \rfloor + 1} + i - \lfloor \lg i \rfloor - 2.$$

Substituting gives the second line:

$$\text{Total}_i \;=\; 3i - (2^{\lfloor \lg i \rfloor + 1} + i - \lfloor \lg i \rfloor - 2).$$

Now split into two cases according to where i sits relative to the powers of two. In both, write $k = \lfloor \lg i \rfloor$.

Case 1: i is exactly a power of two, $i = 2^k$ with $k \geq 0$

Then $\lfloor \lg i \rfloor = k$ exactly, and

$$2^{\lfloor \lg i \rfloor + 1} \;=\; 2^{k+1} \;=\; 2 \cdot 2^k \;=\; 2i.$$

So

$$\begin{aligned} Total_i &= 3i - (2i + i - k - 2) \\ &= 3i - (3i - k - 2) \\ &= k + 2. \end{aligned}$$

That is the first case of **Figure 14.10**: $\text{Total}_i = 3i - (3i - k - 2) = k + 2$.

Since $k \geq 0$, we get $\text{Total}_i \geq 2 > 0$. **The credit is positive.** Good — and note that this is the *tightest* moment, immediately after the most expensive operation the sequence has yet seen. If the balance survives here, it survives everywhere.

Case 2: i lies strictly between powers, $i = 2^k + j$ with $k \geq 0$ and $0 \leq j < 2^k$

Here $\lfloor \lg i \rfloor$ is still k, because $2^k \leq i < 2^{k+1}$. And now

$$2^{\lfloor \lg i \rfloor + 1} \;=\; 2^{k+1} \;=\; 2 \cdot 2^k \;=\; 2\,(i - j),$$

using $2^k = i - j$ from the definition of j. So

$$\begin{aligned} Total_i &= 3i - (2(i - j) + i - k - 2) \\ &= 3i - (2i - 2j + i - k - 2) \\ &= 3i - (3i - 2j - k - 2) \\ &= 2j + k + 2, \end{aligned} \tag{14.9}$$

which is the second case: $\text{Total}_i = 2j + k + 2$.

Since $j \geq 0$ and $k \geq 0$, again $\text{Total}_i \geq 2 > 0$.

(Case 1 is in fact the special case $j = 0$ of Case 2, which is a useful sanity check: putting $j = 0$ into $2j + k + 2$ gives $k + 2$, agreeing with Case 1. It is still worth separating them, because Case 1 is where an expensive operation has just fired and the balance is at its lowest.)

The conclusion

The credit is at least 2 after every operation, so it is certainly non-negative, so the charge of 3 per operation is legitimate. Therefore: *the total amortized cost of* $3n$ *serves as an upper bound on the total actual cost, therefore the complexity of* n *operations is* $O(n)$*; hence the amortized cost per operation is* $O(1)$.

Compare with Section 14.3.4, where the aggregate method gave the bound $3n - \lfloor \lg n \rfloor - 2$. The accounting method gives the slightly weaker $3n$. That is typical: the accounting method trades a little tightness for a much shorter argument, and — more importantly — for a *per-operation* statement rather than a global one.

Checked against a program

Listing 14.5 runs the sequence, tracks the balance, and compares it with Equation (14.9) at every step.

Listing 14.5 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`.

```
// 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.
```

The formula is confirmed for a million operations. Watch the balance: it climbs by 2 on every cheap operation and drops sharply at $i = 4,8,16$ — down to 4, 5 and 6 respectively, which are $k + 2$ for $k = 2,3,4$, exactly as Case 1 says. It never gets near zero, and the margin *grows*: the expensive operations are falling behind the savings, which is why the amortized bound is comfortable rather than tight.

14.6 The Potential Method

14.6.1 The idea

The accounting method stores credit *on objects*: a rupiah on this stack item, a rupiah on that 1 bit. That is concrete and easy to picture, but it becomes awkward when the prepayment is not naturally attached to any single object — when what you have saved up is a property of the structure's *shape* rather than of its contents.

The potential method solves this by storing the prepayment as one number attached to the structure as a whole. In outline:

Each data structure is associated with a non-negative potential. The structure to be used for the n operations is determined in advance. The corresponding sequence of data structures is $D_0, D_1, \ldots, D_n$, with potentials $\Phi(D_0), \Phi(D_1), \ldots, \Phi(D_n)$.

Let us set up the notation with care, because five symbols arrive at once.

THE POTENTIAL METHOD, SET UP. We perform n operations on a data structure.

- D_0 is the structure **before** any operation — the initial state.
- D_i is the structure **after** the i-th operation, for $i = 1,2, \ldots, n$.
- c_i is the **actual cost** of the i-th operation: the real work it does in turning D_{i-1} into D_i.
- Φ (capital phi, a Greek letter) is the **potential function**. It takes a state of the structure and returns a real number, $\Phi(D_i)$. We choose it.
- $\hat{c}_i$ is the **amortized cost** of the i-th operation, defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}). \tag{14.10}$$

Some books write the amortized cost a_i rather than $\hat{c}_i$; they are the same thing, and the definition then reads $a_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

Read definition Equation (14.10) in words. **The amortized cost of an operation is what it actually cost, plus however much it raised the potential** (or minus however much it lowered it). An operation that does little work but leaves the structure in a more dangerous state — more items piled up, more bits set to 1, a fuller table — is charged for the danger it created. An operation that does a great deal of work but *defuses* the structure gets a refund for the danger it removed.

The term $\Phi(D_i) - \Phi(D_{i-1})$ is called the **change in potential**, and is often written $\Delta\Phi_i$.

14.6.2 The telescoping sum

Why does this definition give a valid amortized cost? Add up Equation (14.10) over the whole sequence, as **Figure 14.11** does.

Summing the amortized costs, the potential terms cancel in pairs (telescope):

$$\begin{aligned}\sum_{i=1}^{n} a_i &= \sum_{i=1}^{n}\big(c_i + \Phi(D_i) - \Phi(D_{i-1})\big)\\ &= \Big(\sum_{i=1}^{n} c_i\Big) + \big(\cancel{\Phi(D_1)} - \Phi(D_0)\big) + \big(\cancel{\Phi(D_2)} - \cancel{\Phi(D_1)}\big) + \cdots + \big(\Phi(D_n) - \cancel{\Phi(D_{n-1})}\big)\\ &= \Big(\sum_{i=1}^{n} c_i\Big) + \Phi(D_n) - \Phi(D_0).\end{aligned}$$

So if $\Phi(D_n) \geq \Phi(D_0)$, the total amortized cost $\sum a_i$ is an upper bound on the total actual cost $\sum c_i$.

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.

Here it is with every step written out.

$$\sum_{i=1}^{n} \hat{c}_i \;=\; \sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

Why: this is just the definition of $\hat{c}_i$, substituted inside the sum.

$$\sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) \;=\; \sum_{i=1}^{n} c_i \;+\; \sum_{i=1}^{n}(\Phi(D_i) - \Phi(D_{i-1}))$$

Why: a sum of sums is the sum of the sums; we are just grouping the c terms together and the Φ terms together.

Now look hard at that second sum. Write it out for $n = 4$:

$$(\Phi(D_1) - \Phi(D_0)) + (\Phi(D_2) - \Phi(D_1)) + (\Phi(D_3) - \Phi(D_2)) + (\Phi(D_4) - \Phi(D_3)).$$

The $+\Phi(D_1)$ in the first bracket cancels the $-\Phi(D_1)$ in the second. The $+\Phi(D_2)$ in the second cancels the $-\Phi(D_2)$ in the third. And so on. Everything in the middle cancels, and only the very first negative term and the very last positive term survive:

$$\sum_{i=1}^{n}(\Phi(D_i) - \Phi(D_{i-1})) \;=\; \Phi(D_n) - \Phi(D_0).$$

This is called a **telescoping sum**, after the way an old sailor's telescope collapses into itself.

So

$$\sum_{i=1}^{n}\hat{c}_i \;=\; \sum_{i=1}^{n}c_i \;+\; \Phi(D_n) - \Phi(D_0), \tag{14.11}$$

which is exactly the second line of **Figure 14.11**.

14.6.3 The one condition, and why it is needed

Rearranging Equation (14.11) to isolate the thing we care about:

$$\sum_{i=1}^{n}c_i \;=\; \sum_{i=1}^{n}\hat{c}_i \;-\; \Phi(D_n) \;+\; \Phi(D_0).$$

We want the *real* total on the left to be at most the *invented* total on the right-hand sum. That needs

$$-\Phi(D_n) + \Phi(D_0) \;\leq\; 0,$$

that is,

$$\boxed{\Phi(D_n) \;\geq\; \Phi(D_0)} \tag{14.12}$$

The method is usually stated with condition Equation (14.12) required at every step: $\Phi(D_i) \geq \Phi(D_0)$ *for all i, so that the total potential never drops below its initial value.* Requiring it for all i which is stronger than needed for $i = n$ alone — and, as in Section 14.5.2, the stronger version is the right one, because the sequence may stop anywhere.

> **WHY THE CONDITION IS EXACTLY THE CREDIT INVARIANT IN DISGUISE.** In the accounting method, the credit after k operations was $\sum\hat{c}_i - \sum c_i$, and Equation (14.8) required it to be ≥ 0. Here, by the telescoping identity applied to the first k operations, that same quantity is $\Phi(D_k) - \Phi(D_0)$. So "the credit never goes negative" and "the potential never drops below its starting value" are **the same statement**. The potential *is* the credit; the only difference is that the potential is computed from the state of the structure instead of being tracked object by object.

The practical shortcut. Nearly always we arrange that $\Phi(D_0) = 0$ — the structure starts empty, or at rest — and that $\Phi(D) \geq 0$ for every reachable state D. Then the condition is automatic and nothing further need be checked. Get into the habit of stating both facts explicitly when you use the method; a proof that omits them is incomplete.

How to find Φ. There is no algorithm for this. Φ should measure *how much trouble the structure is storing up* — how far it is from a state in which an expensive operation could not happen. On a stack, trouble is items waiting to be popped. On a counter, trouble is 1 bits waiting to be reset. On a nearly full table, trouble is the coming copy. You guess, you compute the amortized costs, and if they come out constant you have guessed right. Guessing well is a skill that comes from seeing several examples, which is what the rest of this section is for.

14.6.4 The stack, by the potential method

Figure 14.12 summarizes the potential analysis of the stack.

- 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 $k' = \min(k, s)$.

The choice. $\Phi(D)$ = the number of objects currently on the stack.

Check the two requirements. The stack starts empty, so $\Phi(D_0) = 0$. The number of elements on a stack is never negative, so $\Phi(D_i) \geq 0 = \Phi(D_0)$ for every i. Both requirements hold, as **Figure 14.12** records.

Now compute the amortized cost of each operation. Let s be the number of items on the stack *before* the operation, so $\Phi(D_{i-1}) = s$.

`Push`. The actual cost is $c_i = 1$. Afterward the stack holds $s + 1$ items, so $\Phi(D_i) = s + 1$. Therefore

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}) \;=\; 1 + (s+1) - s \;=\; 1 + 1 \;=\; 2.$$

This is the first line of **Figure 14.12**.

`Pop`. The actual cost is $c_i = 1$. Afterward the stack holds $s - 1$ items. Therefore

$$\hat{c}_i \;=\; 1 + (s-1) - s \;=\; 1 - 1 \;=\; 0.$$

`Multipop(S,k)`. Let $k' = \min(k, s)$ — the number of items actually removed. The actual cost is $c_i = k'$. Afterward the stack holds $s - k'$ items. Therefore

$$\hat{c}_i \;=\; k' + (s - k') - s \;=\; k' - k' \;=\; 0,$$

which is the second line of **Figure 14.12**.

So the amortized cost is at most 2 for every operation, and n operations cost at most $2n$ — the same answer for the third time, by the third method. That is the point of doing it three times: the answer is a property of the structure, not of the method.

> **IN PLAIN LANGUAGE.** The potential counts how many items are sitting on the stack waiting to be removed — that is, how much *future work is stored up*. A `Push` does one unit of work now and stores one unit for later, so it is charged 2. A `Multipop` does k' units of work now but

removes k' units of stored future work, so the two cancel and it is charged nothing. The expensive operation is free precisely because it is *spending savings*, not creating obligations.

14.6.5 The binary counter, by the potential method

Figure 14.13 summarizes the potential analysis of the counter.

- Potential: b_i = the number of 1 bits after the i-th `Increment`.
- If the i-th `Increment` resets t_i bits, its actual cost is at most $t_i + 1$, and $b_i \leq b_{i-1} - t_i + 1$.
- Change in potential: $\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$.
- Amortized cost: $a_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$.
- Hence n increments starting from zero cost $O(n)$.

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 $(t_i + 1) + (1 - t_i) = 2$.

The choice. $\Phi(D_i) = b_i$, where b_i is the number of 1 bits in the counter after the i-th `Increment`.

Check the requirements. The counter starts at zero, so $b_0 = 0$ and $\Phi(D_0) = 0$. A count of 1 bits is never negative, so $\Phi(D_i) \geq 0$. Both hold.

The actual cost. Suppose the i-th `Increment` resets t_i bits from 1 to 0. Then it either sets one bit from 0 to 1, or (on overflow) sets none. So the actual cost — the number of writes — is at most

$$c_i \;\leq\; t_i + 1.$$

In words: *the maximum actual cost for that operation is* $t_i + 1$.

The new number of 1 bits. We had b_{i-1} ones. The operation removed t_i of them and added at most one. So

$$b_i \;\leq\; b_{i-1} - t_i + 1,$$

which is the next line of **Figure 14.13**. (It is an inequality rather than an equality only because of the overflow case, where no bit is set and $b_i = b_{i-1} - t_i$.)

The change in potential. Subtract b_{i-1} from both sides:

$$\Phi(D_i) - \Phi(D_{i-1}) \;=\; b_i - b_{i-1} \;\leq\; (b_{i-1} - t_i + 1) - b_{i-1} \;=\; 1 - t_i.$$

The amortized cost. Put the two pieces together:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= t_i + 1 + 1 - t_i \\ &= 2.\end{aligned}$$

The t_i cancels. That cancellation is the whole analysis: **however many bits the increment has to reset, the potential falls by exactly as much, and the two terms annihilate.** So $\hat{c}_i \leq 2$ for every i, and n increments cost at most $2n$ — the same $2n$ as before.

Hence the complexity of n operations is $O(n)$.

Both potentials, watched

Listing 14.6 computes the potential after every operation on both structures, prints c_i, the change in Φ, and $\hat{c}_i$, and finally checks the telescoping identity $\sum \hat{c}_i = \sum c_i + \Phi(D_n) - \Phi(D_0)$ numerically.

Listing 14.6 The potential method instrumented on both the stack and the counter. Compile with `g++ -std=c++17 -O2 -o potential potential.cpp`.

```
// The potential method made visible.  For the stack we take
// Phi(D) = number of objects on the stack; for the counter we take
// Phi(D) = number of 1 bits.  After each operation the program prints
// the actual cost c_i, the change in potential, and the amortized cost
//     a_i = c_i + Phi(D_i) - Phi(D_{i-1}),
// then checks the identity  sum(a_i) = sum(c_i) + Phi(D_n) - Phi(D_0).
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;

struct PotStack {
    vector<int> data;
    long long sumC = 0, sumA = 0;
    long long phi() const { return (long long)data.size(); }

    long long push(int x) { long long b = phi(); data.push_back(x);
                            return acct(1, b); }
    long long pop()       { long long b = phi(); long long c = 0;
                            if (!data.empty()) { data.pop_back(); c = 1; }
                            return acct(c, b); }
    long long multipop(int k) { long long b = phi(); long long c = 0;
                            while (!data.empty() && k != 0) { data.pop_back(); --k; ++c; }
                            return acct(c, b); }
    long long lastC = 0;
private:
    long long acct(long long c, long long before) {
        lastC = c; long long a = c + phi() - before;
        sumC += c; sumA += a; return a;
    }
};

struct PotCounter {
    vector<int> A;
    long long sumC = 0, sumA = 0, lastC = 0, lastT = 0;
    explicit PotCounter(int k) : A(k, 0) {}
    long long phi() const { long long s = 0; for (int b : A) s += b; return s; }
```

Listing 14.6 (Continued) *The potential method instrumented on both the stack and the counter. Compile with g++ -std=c++17 -O2 -o potential potential.cpp.*

```
    long long increment() {
        long long before = phi(), c = 0, t = 0;
        size_t i = 0;
        while (i < A.size() && A[i] == 1) { A[i] = 0; ++c; ++t; ++i; }  // t bits reset
        if (i < A.size()) { A[i] = 1; ++c; }
        lastC = c; lastT = t;
        long long a = c + phi() - before;
        sumC += c; sumA += a; return a;
    }
};

int main() {
    cout << "STACK, Phi = number of objects.  Phi(D_0) = 0.\n";
    cout << "op              size  c_i   Phi before  Phi after   a_i\n";
    PotStack S;
    auto line = [&](const string& w, long long a, long long before) {
        cout << left << setw(16) << w << right << setw(4) << S.data.size()
             << setw(6) << S.lastC << setw(12) << before
             << setw(11) << S.phi() << setw(7) << a << "\n";
    };
    { long long b = S.phi(); long long a = S.push(1);   line("PUSH", a, b); }
    { long long b = S.phi(); long long a = S.push(2);   line("PUSH", a, b); }
    { long long b = S.phi(); long long a = S.push(3);   line("PUSH", a, b); }
    { long long b = S.phi(); long long a = S.push(4);   line("PUSH", a, b); }
    { long long b = S.phi(); long long a = S.push(5);   line("PUSH", a, b); }
    { long long b = S.phi(); long long a = S.pop();     line("POP", a, b); }
    { long long b = S.phi(); long long a = S.multipop(3); line("MULTIPOP 3", a, b); }
    { long long b = S.phi(); long long a = S.multipop(9); line("MULTIPOP 9", a, b); }
    cout << "sum c_i = " << S.sumC << ", sum a_i = " << S.sumA
         << ", Phi(D_n) - Phi(D_0) = " << S.phi() - 0
         << "; identity holds: "
         << ((S.sumA == S.sumC + S.phi()) ? "yes" : "no") << "\n";

    cout << "\nCOUNTER, Phi = number of 1 bits.  Phi(D_0) = 0.\n";
    cout << "  n  value  t_i (bits reset)  c_i   Phi after   a_i\n";
    PotCounter C(8);
    for (int n = 1; n <= 16; ++n) {
        long long a = C.increment();
        cout << setw(3) << n << setw(7) << n << setw(15) << C.lastT
             << setw(7) << C.lastC << setw(11) << C.phi() << setw(7) << a << "\n";
    }
    cout << "sum c_i = " << C.sumC << ", sum a_i = " << C.sumA
         << " (= 2n = " << 2 * 16 << "), Phi(D_n) = " << C.phi() << "\n";
    cout << "every a_i above is exactly 2, which is the bound proved in the text\n";
    return 0;
}
```

```
STACK, Phi = number of objects.  Phi(D_0) = 0.
op              size  c_i   Phi before  Phi after   a_i
PUSH               1     1           0          1      2
PUSH               2     1           1          2      2
PUSH               3     1           2          3      2
PUSH               4     1           3          4      2
PUSH               5     1           4          5      2
POP                4     1           5          4      0
MULTIPOP 3         1     3           4          1      0
MULTIPOP 9         0     1           1          0      0
sum c_i = 10, sum a_i = 10, Phi(D_n) - Phi(D_0) = 0; identity holds: yes

COUNTER, Phi = number of 1 bits.  Phi(D_0) = 0.
  n  value  t_i (bits reset)  c_i   Phi after   a_i
  1      1              0      1          1      2
  2      2              1      2          1      2
  3      3              0      1          2      2
  4      4              2      3          1      2
  5      5              0      1          2      2
  6      6              1      2          2      2
  7      7              0      1          3      2
  8      8              3      4          1      2
  9      9              0      1          2      2
 10     10              1      2          2      2
 11     11              0      1          3      2
 12     12              2      3          2      2
 13     13              0      1          3      2
 14     14              1      2          3      2
 15     15              0      1          4      2
 16     16              4      5          1      2
sum c_i = 31, sum a_i = 32 (= 2n = 32), Phi(D_n) = 1
every a_i above is exactly 2, which is the bound proved in the text
```

The counter table is worth studying. The column c_i leaps around — 1, 2, 1, 3, 1, 2, 1, 4, ..., 5 — and the column a_i is a flat wall of 2s. The t_i column shows why: whenever c_i is large, t_i is large too, and t_i is exactly what the potential drop cancels. And the totals confirm the telescoping identity: $\sum c_i = 31$, $\Phi(D_n) = 1$, and $31 + 1 = 32 = \sum \hat{c}_i$.

The stack run does the same thing from the other side: the `Multipop 3` genuinely costs 3, and is charged 0, because the potential fell from 4 to 1.

14.6.6 Exercise 17.3-6: a queue from two stacks

This is the best exercise of the chapter, because the potential function is not obvious and the structure is one you may actually want to build.

> **EXERCISE 17.3-6.** Show how to implement a queue with two ordinary stacks so that the amortized cost of each `Enqueue` and each `Dequeue` operation is $O(1)$.

The problem

A **stack** is last-in-first-out: the item you get back is the one you put in most recently. A **queue** is first-in-first-out: the item you get back is the one that has been waiting longest — like a line at a bank. Suppose all you are given is stacks. Can you build a queue?

Yes, and the trick is that reversing a stack turns it into a queue. If you pop everything off stack A and push it onto stack B as you go, then the item at the *bottom* of A — the oldest — ends up at the *top* of B, where it can be popped first.

The design

The design takes three sentences:

To implement a queue using stacks, two stacks are used: one as the input stack (S_{IN}) and the other as the output stack (S_{OUT}).

For an `Enqueue` operation, the element is pushed onto the input stack (S_{IN}).

For a `Dequeue` operation, an element is popped from the output stack (S_{OUT}). If S_{OUT} is empty, all elements from S_{IN} are first transferred to S_{OUT} (by popping from S_{IN} and pushing onto S_{OUT}), after which one element is popped from S_{OUT}.

Figure 14.14 gives the pseudocode.

```
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 of the pseudocode deserve a comment, because getting them wrong is the classic bug.

The transfer happens only when S_{OUT} is empty. If you transferred on every `Dequeue`, you would reverse the order over and over and the structure would be both wrong and slow. The whole amortized argument depends on the transfer being rare.

The underflow test comes after the transfer. A queue is empty only when *both* stacks are empty. Testing S_{OUT} before the transfer would wrongly report underflow whenever the items happen to be sitting in S_{IN}.

Why it is correct

Items enter S_{IN} in arrival order, so the oldest item in S_{IN} is at the bottom. A transfer pops them off one at a time — newest first — and pushes them onto S_{OUT}, so the oldest ends up on top of S_{OUT}. While S_{OUT} is non-empty, `Dequeue` serves from its top, which is always the oldest item still in the structure. New arrivals go into S_{IN} and cannot jump the queue, because they will only reach S_{OUT} after it has been completely drained. So the structure really is first-in-first-out.

Why the naive cost bound is bad

A `Dequeue` that triggers a transfer moves every item in S_{IN}. If the queue holds m items that could be $\Theta(m)$ work, so the worst case for a single `Dequeue` is $\Theta(n)$ in a sequence of n operations, and the crude bound is $O(n^2)$ — the same wrong answer as always.

The potential function

Figure 14.15 gives the complete potential analysis.

- 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.

The choice. $\Phi(D_i) = 2 \times \text{elements}_i$, where elements_i is the number of elements **in the input stack** S_{IN} after the i-th operation. Note carefully: items sitting in S_{OUT} contribute *nothing*.

Why that is the right guess. The expensive event is a transfer, and a transfer touches exactly the items in S_{IN} — twice each, once to pop and once to push. Items in S_{OUT} are never going to be transferred again; they have already paid. So the "stored-up trouble" in the structure is exactly two units per item waiting in S_{IN}, and that is the potential.

Check the requirements. Both stacks start empty, so $\Phi(D_0) = 0$. A count of elements is never negative, so $\Phi(D_i) \geq 0$. Both hold.

Now the three cases, following **Figure 14.15** line by line. Write e for elements_{i-1}, the size of S_{IN} before the operation.

Case 1: Enqueue. One push onto S_{IN}: actual cost $c_i = 1$. Afterward S_{IN} holds $e + 1$ items, so the potential goes from $2e$ to $2(e + 1)$, a rise of 2.

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}) \;=\; 1 + 2 \;=\; 3.$$

This is the first line of **Figure 14.15**.

Case 2: Dequeue with S_{OUT} non-empty — no transfer needed. One pop from S_{OUT}: actual cost $c_i = 1$. S_{IN} is untouched, so the potential does not change at all.

$$\hat{c}_i \;=\; 1 + 0 \;=\; 1.$$

This is the second line.

Case 3: Dequeue with S_{OUT} empty — a transfer happens. This is the expensive one. The transfer moves all e items out of S_{IN} and into S_{OUT}. Each item costs 2 to move: one pop and one push. Then one final pop from S_{OUT} returns the answer. So

$$c_i \;=\; 2e + 1.$$

Afterward S_{IN} is empty, so $\Phi(D_i) = 0$, while $\Phi(D_{i-1}) = 2e$.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= (2e+1) + 0 - 2e \\ &= 1.\end{aligned}$$

This is the third line, written in the figure as $(2 \times \text{elements}_{i-1} + 1) + 0 - 2 \times \text{elements}_{i-1} = 1$.

The conclusion. Every operation has amortized cost at most 3. So n operations cost at most $3n = O(n)$, and the amortized cost per operation is $O(1)$: *the amortized cost for both* `Enqueue` *and* `Dequeue` *operations is* $O(1)$.

> **IN PLAIN LANGUAGE.** Every item makes exactly one journey: pushed onto S_{IN}, popped off S_{IN}, pushed onto S_{OUT}, popped off S_{OUT}. Four steps, once each, in its entire lifetime. `Enqueue` pays for three of them in advance — one for the push it performs and two banked for the transfer — and the fourth, the last pop, is paid by the charge of 1 on the `Dequeue` that removes it. A `Dequeue` that finds a thousand items waiting does a thousand units of work and is charged 1, because those thousand items each left two units of credit behind when they arrived.

The program

Listing 14.7 implements the pseudocode of **Figure 14.14** exactly, computes Φ after each operation, and prints the actual and amortized cost side by side.

Listing 14.7 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`.

```
// 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 14.7 (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;
}
```

```
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.
```

Read the `returned` column first: 1, 2, 3, 4, 5, 6 — the queue really is first-in-first-out. Now read the `actual` column against the `amortized` column. The sixth line is the expensive `Dequeue`: it transfers five items at a cost of 2 each plus the closing pop, so $c_i = 11$ — and its amortized cost is 1, because the potential fell from 10 to 0. Every `Enqueue` is charged exactly 3 and every `Dequeue` exactly 1, precisely as the three cases predict.

The long run makes the point at scale: one operation costs 2001 units, and the average over 200,000 operations is 2.

14.7 Dynamic Tables

14.7.1 The problem

Everything so far has been a warm-up for this. The dynamic table is the structure you use every day without noticing: it is what `std::vector` is, what Python's `list` is, what Java's `ArrayList` is — the resizing array of Sedgewick and Wayne [5]. Understanding its amortized cost is the difference between writing `v.push_back(x)` in a loop with confidence and writing it with vague unease.

The goal can be stated in two sentences:

Using amortized analysis, it will be demonstrated that the amortized cost of inserting and deleting items in the table is $O(1)$, even though the actual cost of these operations may be higher when they trigger an expansion or contraction. Furthermore, it will be shown that the amount of unused space in the table never exceeds a certain constant factor relative to the total allocated space.

Two goals, then. **Speed**: $O(1)$ amortized per operation. **Space**: never waste more than a constant fraction of what we allocated. We must hit both; either one alone is easy.

Here is the situation. We want a table — an array — that holds an unknown number of items. We cannot ask the operating system for "an array that grows", because memory is allocated in fixed blocks. What we can do is allocate a block of some size, fill it, and when it is full, allocate a bigger block, copy everything across, and free the old one.

The vocabulary is:

- `TABLE-INSERT` inserts an item into a slot.
- `TABLE-DELETE` removes an item from the table and frees its slot.
- $num[T]$ is the number of items currently stored, $size[T]$ the number of slots allocated.
- The **load factor** $\alpha(T)$ is $num[T]/size[T]$ — the fraction of the allocated space actually in use. (By convention, an empty table with $size[T] = 0$ has $\alpha(T) = 1$, so that the "table is full" test works from the very beginning.)

> **A CLASH OF NOTATION.** This section uses α for the load factor, and Section 14.8 will use α for the inverse Ackermann function. They are unrelated. This is unfortunate but it is the notation both literatures use, so you must live with it; context always makes clear which is meant. In this section, α is a fraction between 0 and 1.

The first policy to analyze is this: *if all slots are filled, that is, if $\alpha(T)$ equals 1, the table is expanded to twice the number of slots of the old table.* **Doubling.**

14.7.2 `TABLE-INSERT`

Figure 14.16 gives the pseudocode.

```
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.

Read it carefully.

Lines 1–3. If the table has no slots at all, allocate one slot. This is the bootstrap case: you cannot double zero.

Line 4. Test whether the table is full: $num[T] = size[T]$.

Lines 5–9. If it is full: allocate a new block with $2 \cdot size[T]$ slots; copy every item from the old block into the new one; free the old block; make $table[T]$ point at the new block; and record the new size.

Lines 10–11. Insert x into whatever block $table[T]$ now points at, and increase $num[T]$ by one.

The cost. We count one unit per **elementary insertion**: one unit to write an item into a slot. So:

- If there was room, the cost is exactly 1 — line 10 writes one item.
- If an expansion happened, line 6 copies $num[T]$ items (which equals the old $size[T]$, since the table was full) and then line 10 writes one more. So the cost is $num[T] + 1$, and since $num[T]$ was $size[T]$ and after the insert the table holds i items, the cost is exactly i where i is the index of this insertion.

In one sentence: *the cost of the i-th operation, if space is still available in the table, is $c_i = 1$. If the table is full, an expansion will occur, resulting in a cost of $c_i = i$.*

When does an expansion happen? The table is full at sizes 1,2,4,8,16, … — it doubles each time. So the insertions that trigger an expansion are the 2nd (table of size 1 is full), the 3rd (size 2 is full), the 5th (size 4 is full), the 9th, the 17th, … The pattern is: **insertion i triggers an expansion exactly when $i - 1$ is an exact power of 2** (taking the 1st insertion as the special bootstrap case). Call this the **expansion condition**.

Check it: $i = 2 \Rightarrow i - 1 = 1 = 2^0$ ✓; $i = 3 \Rightarrow 2 = 2^1$ ✓; $i = 5 \Rightarrow 4 = 2^2$ ✓; $i = 9 \Rightarrow 8 = 2^3$ ✓; $i = 4 \Rightarrow 3$, not a power of 2, so no expansion ✓.

The crude bound. One insertion can cost n, there are n of them, total $O(n^2)$. As always, we can do much better.

14.7.3 The aggregate method applied to `TABLE-INSERT`

Figure 14.17 does it in one display.

$$\begin{aligned}\sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n\end{aligned}$$

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 $3n$.

Here is the derivation, expanded.

The baseline. Every insertion writes at least one item, so n insertions contribute at least n. Write that as the n on the right of the inequality.

The expansions. When insertion i triggers an expansion, the extra cost beyond the baseline 1 is the number of items copied, which is $i - 1$ — and $i - 1$ is a power of 2, say 2^j. So the expansion that happens at insertion $2^j + 1$ costs an extra 2^j.

Which j occur? Those with $2^j + 1 \le n$, i.e. $2^j \le n - 1 < n$, i.e. $j \le \lfloor \lg n \rfloor$.

Therefore

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j, \tag{14.13}$$

which is the first line of **Figure 14.17**.

Evaluate the geometric sum in Equation (14.13) with formula Equation (14.6) of Section 14.3.4, $\sum_{k=0}^{m} x^k = (x^{m+1} - 1)/(x - 1)$, with $x = 2$ and $m = \lfloor \lg n \rfloor$:

$$\sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \;=\; 2^{\lfloor \lg n \rfloor + 1} - 1 \;<\; 2^{\lg n + 1} \;=\; 2n.$$

So

$$\sum_{i=1}^{n} c_i \;<\; n + 2n \;=\; 3n,$$

the second and third lines of **Figure 14.17**. Hence the total cost of n insertions is $O(n)$, and the amortized cost per insertion is

$$\frac{3n}{n} \;=\; 3 \;=\; O(1).$$

In short: *for n operations, the worst-case complexity is $O(n)$; the amortized cost per operation is $O(1)$.*

> **WHY DOUBLING AND NOT ADDING.** The whole argument rests on the expansion costs forming a *geometric* series: 1,2,4,8, …, each twice the last. A geometric series that doubles sums to less than twice its largest term, so the sum of *all* the copying ever done is less than twice the size of the table at the end. If instead you grew the table by a fixed amount — say 100 slots each time — the expansion costs would be 100,200,300, …, an *arithmetic* series, whose sum is $\Theta(n^2/200)$. Exercise 14.10 and **Listing 14.13** make this concrete. **The doubling is not an implementation detail; it is the whole idea.**

14.7.4 The accounting method applied to `TABLE-INSERT`

The accounting argument is beautifully compact:

Using the Accounting Method, a single `TABLE-INSERT` operation has an amortized cost of 3. Each item pays for three insertion-related events:

1. the insertion of the item itself into the table,
2. the relocation of the item when the table is expanded,
3. the relocation of another item that has previously been moved once.

Let us make that concrete, because the third clause is the clever one and it is easy to skim past.

Charge every insertion 3 units. One is spent immediately writing the item into its slot. Two are banked *on that item*.

Now consider the next expansion. The table was full at size s and doubles to $2s$. All s items must be copied. Who pays?

Look at when the table reached size s. Immediately after *that* expansion, the table held $s/2$ items and had s slots. So $s/2$ of the items present at the next expansion are **old** — they were there before — and $s/2$ are **new**, inserted since. Only the new ones still have their savings; the old ones spent theirs at the previous expansion.

Each new item carries 2 banked units. There are $s/2$ of them, so they have $2 \times (s/2) = s$ units between them — exactly what is needed to copy all s items. So each new item pays for **itself** being copied (clause 2) and for **one old item** being copied (clause 3). That is the third event in the list above, and that is why the charge is 3 rather than 2.

The credit invariant. After any prefix of insertions, the credit stored in the table is $2 \times$ (the number of items inserted since the last expansion), which is a count and so cannot be negative. Hence the credit never goes negative, hence the charge of 3 is legitimate, hence n insertions cost at most $3n$.

So for n operations the worst-case complexity is $O(n)$, and the amortized cost per operation is $O(1)$. Same bound, second method.

14.7.5 The potential method applied to `TABLE-INSERT`

The potential method makes this choice:

$$\Phi(T) \;=\; 2 \cdot num[T] \;-\; size[T]. \tag{14.14}$$

Potential Equation (14.14) is one of the most famous potential functions in the subject, and it repays a minute's thought about *why* it is right.

Reading it. Immediately after an expansion the table is half full: $num = size/2$, so $\Phi = 2(size/2) - size = 0$. The table is in its safest state and its potential is zero. As items are inserted, num climbs while $size$ stays put, so Φ climbs — by exactly 2 per insertion. When the table becomes full, $num = size$ and $\Phi = 2\,size - size = size$: the potential now equals the number of items that the next expansion will have to copy. **The potential has saved up exactly the cost of the disaster it can see coming.**

Check the requirements. *The initial value of this potential function is 0* — because an empty table has $num = size = 0$. *Since the table always contains at least half of its capacity, i.e.* $num[T] \geq size[T]/2$, *it follows that* $\Phi(T)$ *is always non-negative.* And why is the table always at least half full? Because it was exactly half full immediately after the last expansion (we doubled from full), and insertions only raise num. So $2\,num \geq size$, i.e. $\Phi \geq 0$. Both requirements hold.

Now the two cases, summarized in **Figure 14.18**. Write num_i and $size_i$ for the values after the i-th insertion.

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 $num_i - 3$.

Case 1: no expansion. Then $size_i = size_{i-1}$ and $num_i = num_{i-1} + 1$, and the actual cost is $c_i = 1$.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (2\,num_i - size_i) - (2\,num_{i-1} - size_{i-1}) \\ &= 1 + 2\,num_i - size_i - 2(num_i - 1) + size_i \\ &= 1 + 2\,num_i - 2\,num_i + 2 - size_i + size_i \\ &= 1 + 2 \\ &= 3.\end{aligned}$$

In one line: $\hat{c}_i = 1 + (2\,num_i - size_i) - (2\,num_{i-1} - size_{i-1}) = 3$.

Case 2: an expansion happens. Two relationships hold:

$$\frac{size_i}{2} \;=\; size_{i-1} \;=\; num_i - 1.$$

Read that as two facts. *First*, $size_i = 2\,size_{i-1}$ — we doubled. *Second*, $size_{i-1} = num_i - 1$ — the table was full before this insertion, so it held $size_{i-1}$ items, and after the insertion it holds one more, namely num_i. Also $num_{i-1} = num_i - 1$.

The actual cost is $c_i = num_i$: we copy the $num_i - 1$ items that were there, and write the new one.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= num_i + (2\,num_i - size_i) - (2\,num_{i-1} - size_{i-1}) \\ &= num_i + 2\,num_i - 2\,size_{i-1} - 2(num_i - 1) + size_{i-1} \\ &= num_i + 2\,num_i - 2(num_i - 1) - 2\,num_i + 2 + (num_i - 1) \\ &= num_i + 2\,num_i - 2\,num_i + 2 - 2\,num_i + 2 + num_i - 1 \\ &= 3.\end{aligned}$$

(In the fourth line both $size_i = 2\,size_{i-1}$ and $size_{i-1} = num_i - 1$ have been substituted. Collect the num_i terms: $1 + 2 - 2 - 2 + 1 = 0$. Collect the constants: $2 + 2 - 1 = 3$.)

Both cases give exactly **3**. That is not an upper bound with slack in it; it is an equality. The amortized cost of the `TABLE-INSERT` operation is therefore $O(1)$.

The picture

Figure 14.19 plots all three quantities over a run of 32 insertions, and it is the single most useful picture in this chapter.

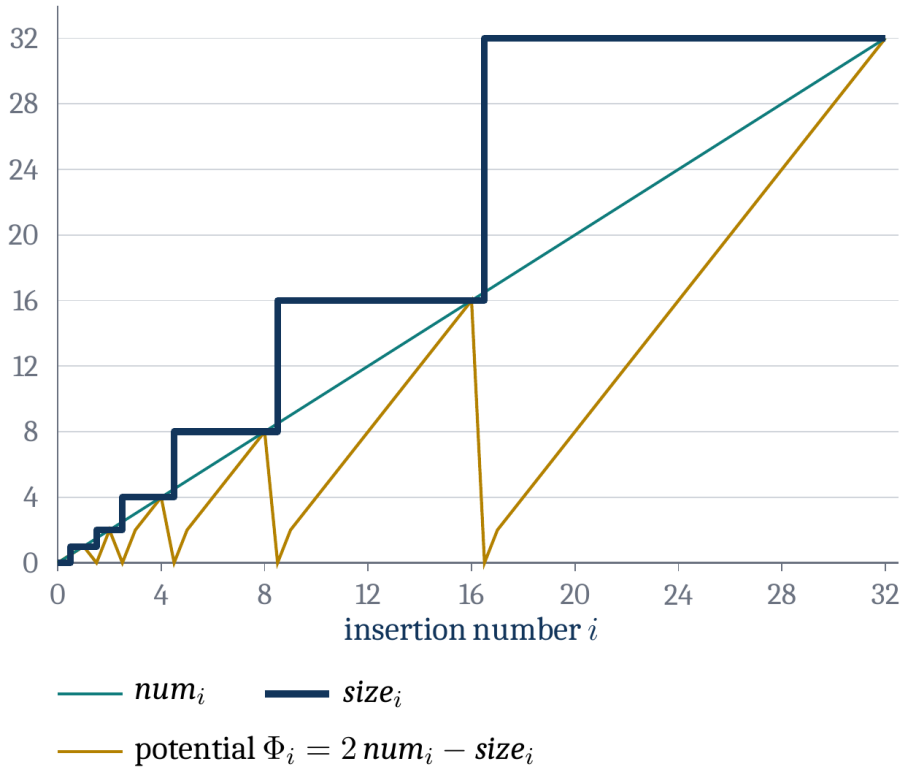


Figure 14.19 A sequence of insertions: the thin diagonal is num, the heavy staircase is $size$, and the sawtooth is $\Phi = 2\,num - size$, which falls to 0 at each expansion and climbs at slope 2 until it meets $size$ again.

Trace the sawtooth with your finger. Just before an expansion, $num = size$ and $\Phi = size$ — the potential has climbed all the way up to the height of the staircase. The expansion fires: $size$ jumps to $2\,size$, and Φ drops instantly to 0, because the newly doubled table is exactly half full. Then the climb begins again, at slope 2, because each insertion adds 1 to num and hence 2 to $2\,num$.

The picture is the proof. The potential is the savings account. It fills up steadily while nothing dramatic is happening, and it is emptied in a single instant by the expansion — which is exactly what savings are for.

The program

Listing 14.8 implements `TABLE-INSERT` with real `malloc`, `memcpy` and `free`, so the copying is real work and not a simulation, and prints c_i, Φ before, Φ after, and $\hat{c}_i$ for the first seventeen insertions.

Listing 14.8 `TABLE-INSERT` with doubling, instrumented with the potential function. Compile with `g++ -std=c++17 -O2 -o dyntable dyntable.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 14.8 (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;
}
```

```
 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 a_i` column is a wall of 3s — the derivation said it would be exactly 3, and it is exactly 3, at every step except the bootstrap insertion where there was no table to copy. The `actual c_i` column is 1,2,3,1,5,1,1,1,9, … ,17: expensive exactly at $i = 2,3,5,9,17$, which are $1 + 2^j$ as predicted. Watch the potential at $i = 16$: it reaches 16, exactly the number of items the next expansion must copy — and at $i = 17$ it collapses to 2.

A million insertions cost 2,048,575 units against the proved ceiling of 3,000,000, and perform only 20 expansions. Twenty. Doubling is extremely effective.

14.7.6 Deletion, and the temptation to halve at half

Now add `TABLE-DELETE`, which removes an item and frees its slot. If we never shrink the table, the amortized cost of a deletion is trivially $O(1)$ — but then a table that grew to a million entries and then had 999,999 of them deleted would still be holding a million slots of memory. That violates the second goal of Section 14.7.1: *the amount of unused space in the table never exceeds a certain constant factor relative to the total allocated space.*

So we must contract. The obvious policy is the mirror image of expansion:

Policy A. When a deletion leaves the table less than half full, halve its size: allocate a block of $size/2$ slots, copy the num remaining items across, and free the old block.

It looks right. It is wrong, and the way it is wrong is worth understanding in detail, because it is the standard trap and it is the reason real implementations use a different constant.

The thrashing sequence, written out

Here is a sequence of operations that destroys Policy A. Suppose the table has just become exactly full: $num = size = s$, where s is a power of two. Now perform, over and over, the four-operation block

$$\text{INSERT}, \quad \text{DELETE}, \quad \text{DELETE}, \quad \text{INSERT}.$$

Follow it one operation at a time.

1. `INSERT`. The table is full ($num = size = s$), so it expands. New size $2s$; s items copied; one written. Cost $s + 1$. State: $num = s + 1$, $size = 2s$.

2. `DELETE`. Remove one item. Now $num = s$ and $size = 2s$, so the load factor is exactly $1/2$. "Less than half full" is **false** — s is not less than s — so no contraction. Cost 1. State: $num = s$, $size = 2s$.

3. `DELETE`. Remove another. Now $num = s - 1$ and $size = 2s$, and $s - 1 < s$, so the table *is* less than half full. Contract: allocate s slots, copy the $s - 1$ remaining items. Cost $1 + (s - 1) = s$. State: $num = s - 1$, $size = s$.

4. `INSERT`. $num = s - 1$ is less than $size = s$, so there is room. Cost 1. State: $num = s$, $size = s$ — **exactly where we started**.

The block is a cycle. It costs

$$(s + 1) + 1 + s + 1 \;=\; 2s + 3$$

and consists of four operations. So the cost per operation is about $s/2$.

Now make s grow with the problem. Take a sequence of n operations: use the first $n/4$ to fill a table to $num = size = n/4$, then use the remaining $3n/4$ operations to repeat the block. Each block of 4 costs about $2 \times (n/4) = n/2$, and there are about $3n/16$ blocks, so the total is about

$$\frac{3n}{16} \times \frac{n}{2} \;=\; \frac{3n^2}{32} \;=\; \Theta(n^2).$$

Policy A is $\Theta(n)$ amortized per operation, not $O(1)$. The crude bound was not pessimistic after all — for this policy it is achieved.

WHY IT FAILS: THE STRUCTURE HAS NO SLACK. After an expansion, the table sits at load factor exactly 1/2 — which is the *same* point at which contraction triggers. One deletion is enough to push it over the edge, and one insertion is enough to push it back. The structure is balanced on a knife edge, and every operation tips it. The expansion and the contraction are each doing $\Theta(s)$ work, and neither has been paid for, because there was no run of cheap operations in between to do the paying. **An amortized bound needs room between the trigger points.**

The fix

Policy B. Expand (double) when the table becomes full, as before. Contract (halve) only when a deletion leaves the table **less than a quarter full**.

Now there is slack. Immediately after an expansion the load factor is 1/2; the contraction trigger is at 1/4. To get from one to the other you must delete a quarter of the table's capacity — that is, $size/4$ operations must go by before a contraction can fire. Immediately after a contraction the load factor is 1/2 again (we halved a quarter-full table, so $num = size_{\text{new}}/2$); to reach the expansion trigger at load factor 1 you must insert another $size/2$ items.

In both directions, a $\Theta(size)$ event can only happen after $\Theta(size)$ cheap operations have paid for it. That is exactly the condition an amortized bound needs.

The load factor is now confined: it never rises above 1 and never falls below 1/4 (except for a table so small it has one slot). So at least a quarter of the allocated space is always in use, and **the wasted space never exceeds three quarters of the allocation** — a constant factor, which is the second goal of Section 14.7.1, achieved.

Measuring both policies

Listing 14.9 implements both policies and runs the thrashing sequence on each, at five different scales.

Listing 14.9 The two contraction policies compared on the thrashing sequence. Compile with `g++ -std=c++17 -O2 -o dyntable2 dyntable2.cpp`.

```
// Dynamic tables that shrink as well as grow.  Two contraction policies
// run on exactly the same sequence:
//   POLICY A -- halve the table the moment it is less than half full;
//   POLICY B -- halve the table only when it is less than a quarter full.
// The sequence is the thrashing sequence of the text: fill the table to
// exactly full, then repeat the block  INSERT, DELETE, DELETE, INSERT.
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

struct Table {
    long long num = 0, size = 0;
    long long actual = 0, ops = 0, expansions = 0, contractions = 0, worstOp = 0;
    int shrinkDenominator;                // 2 for policy A, 4 for policy B

    explicit Table(int d) : shrinkDenominator(d) {}

    void charge(long long c) { actual += c; ++ops; if (c > worstOp) worstOp = c; }

    void insert() {
        long long c = 0;
        if (size == 0) size = 1;
        if (num == size) { c += size; size *= 2; ++expansions; }  // copy `size` items
        ++num; c += 1;
        charge(c);
    }

    void remove() {
        if (num == 0) { charge(0); return; }
        long long c = 1;                                          // free one slot
        --num;
        if (size > 1 && num * shrinkDenominator < size) {
            c += num;                                             // copy `num` items
            size /= 2; ++contractions;
        }
        charge(c);
    }
```

Listing 14.9 (Continued) *The two contraction policies compared on the thrashing sequence. Compile with g++ -std=c++17 -O2 -o dyntable2 dyntable2.cpp.*

```
    // The potential of the combined case, as derived in the text.
    long long potential() const {
        if (size == 0) return 0;
        if (2 * num >= size) return 2 * num - size;
        return size / 2 - num;
    }
};

void run(const string& name, int denom, long long full, long long blocks) {
    Table T(denom);
    while (T.num < full) T.insert();          // now num == size == full
    long long c0 = T.actual, o0 = T.ops, e0 = T.expansions, k0 = T.contractions;
    for (long long b = 0; b < blocks; ++b) { T.insert(); T.remove(); T.remove();
T.insert(); }
    long long cost = T.actual - c0, nops = T.ops - o0;
    cout << left << setw(22) << name << right
         << " table=" << setw(8) << full
         << " ops=" << setw(9) << nops
         << " cost=" << setw(12) << cost
         << " cost/op=" << setw(11) << fixed << setprecision(2)
         << (double)cost / (double)nops
         << "  expansions=" << setw(8) << T.expansions - e0
         << " contractions=" << T.contractions - k0 << "\n";
}

int main() {
    cout << "THRASHING TEST.  Fill the table to exactly full, then repeat the\n"
            "block  INSERT, DELETE, DELETE, INSERT  until n/2 operations are done.\n\n";
    for (long long n : {1024LL, 4096LL, 16384LL, 65536LL, 262144LL}) {
        long long full = n / 4, blocks = (n / 2) / 4;
        cout << "n = " << setw(7) << n << "\n";
        run("  A: shrink below 1/2", 2, full, blocks);
        run("  B: shrink below 1/4", 4, full, blocks);
    }

    cout << "\nPotential and worst case on a long mixed run "
            "(400000 inserts, then 400000 deletes):\n";
    for (int denom : {2, 4}) {
        Table T(denom);
        long long lowest = 0;
        for (int i = 0; i < 400000; ++i) { T.insert(); if (T.potential() < lowest) lowest =
T.potential(); }
        for (int i = 0; i < 400000; ++i) { T.remove(); if (T.potential() < lowest) lowest =
T.potential(); }
        cout << "  shrink below 1/" << denom
             << "   cost=" << setw(9) << T.actual
             << "  cost/op=" << fixed << setprecision(4)
             << (double)T.actual / (double)T.ops
             << "  worst single op=" << setw(8) << T.worstOp
```

Listing 14.9 (Continued) *The two contraction policies compared on the thrashing sequence. Compile with g++ -std=c++17 -O2 -o dyntable2 dyntable2.cpp.*

```
             << "  lowest potential=" << lowest
             << "  final num=" << T.num << " size=" << T.size << "\n";
    }
    return 0;
}
```

```
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
```

This is the clearest experiment in the chapter. Read the `cost/op` column for Policy A down the page: 128.75, 512.75, 2048.75, 8192.75, 32768.75. Each time n is multiplied by 4, the cost per operation is multiplied by 4. That is linear growth in n: the amortized cost per operation of Policy A is $\Theta(n)$, exactly as the hand analysis predicted. Note also the `contractions` column — Policy A performs 32,768 contractions on the largest run, one in every block, each copying tens of thousands of items.

Now read Policy B: 1.50 at every scale. Constant. It performs **one** expansion in the entire run and **zero** contractions, because the four-operation block never drives the load factor down as far as 1/4.

> **THE MORAL, WHICH APPLIES FAR BEYOND TABLES.** When you build a structure that reorganizes itself at a threshold, **the threshold for growing and the threshold for shrinking must not be the same point.** Leave a gap — hysteresis, an engineer would call it — so that the structure must be pushed a long way before it reorganizes again. Thermostats work on the same principle, which is why your air conditioner does not switch on and off once a second.

14.7.7 The potential function for the combined case

For the record, here is the potential function that proves Policy B is $O(1)$ amortized. It is the one implemented in `Table::potential()` in **Listing 14.9**. Let $\alpha = num/size$ be the load factor. Define

$$\Phi(T) = \begin{cases} 2\,num[T] - size[T] & \text{if } \alpha(T) \geq \frac{1}{2}, \\ \frac{size[T]}{2} - num[T] & \text{if } \alpha(T) < \frac{1}{2}. \end{cases} \tag{14.15}$$

Three things to notice about Equation (14.15).

It is zero when the table is exactly half full. Put $num = size/2$ into either branch: the first gives $2(size/2) - size = 0$, the second gives $size/2 - size/2 = 0$. So the two branches agree at the join, and the function has no jump.

It is non-negative everywhere. In the upper branch, $\alpha \geq 1/2$ means $2\,num \geq size$, so $2\,num - size \geq 0$. In the lower branch, $\alpha < 1/2$ means $num < size/2$, so $size/2 - num > 0$.

It rises toward both disasters. As the table fills toward $\alpha = 1$, the first branch climbs to $size$ — the cost of the expansion. As the table empties toward $\alpha = 1/4$, the second branch climbs to $size/2 - size/4 = size/4$ — the cost of the contraction, which copies the $size/4$ surviving items. In both directions the potential saves up exactly what the coming event will cost.

The full case analysis is long — there are four cases for insertion (expansion or not, crossing the half-full line or not) and four for deletion — and it is set as Exercise 14.12 with a complete solution. The headline result, which we quote and which the measurement above supports, is:

> **THEOREM (dynamic tables).** In a dynamic table that doubles when it becomes full and halves when it falls below quarter-full, starting from an empty table, any sequence of n `TABLE-INSERT` and `TABLE-DELETE` operations takes $O(n)$ time in total. The amortized cost of each operation is $O(1)$, and the load factor never falls below $1/4$, so no more than three quarters of the allocated space is ever wasted.

That is both goals of Section 14.7.1, delivered.

14.8 Union–Find: The Promise from Chapter 11, Kept

14.8.1 What we are analyzing

Chapter 11 built a **disjoint-set forest** — an idea going back to Galler and Fisher [6], and explained gently by Kleinberg and Tardos [7] — to make Kruskal's algorithm efficient. Recall the structure, briefly, so this section stands on its own.

We have n elements. They are partitioned into disjoint sets. Each set is stored as a tree, and the root of the tree is the set's **representative**. Each element x has one pointer, $parent[x]$; a root points at itself. Three operations:

- `MakeSet(x)` — create a one-element set: $parent[x] \leftarrow x$.
- `FindSet(x)` — return the representative of x's set, by following parent pointers up to the root.
- `Union(x,y)` — merge the two sets containing x and y, by making one root point at the other.

The cost of `FindSet(x)` is the number of pointers it follows, which is the depth of x in its tree. So the whole question is: **how deep can the trees get?**

Without care, very deep. If you unite $\{1\}$ with $\{2\}$, then the result with $\{3\}$, then with $\{4\}$, always pointing the old root at the new element, you build a path of length $n - 1$, and each `FindSet` on the bottom element costs $\Theta(n)$.

14.8.2 Trick 1: union by rank

Keep, with each root x, an integer $rank[x]$ — an upper bound on the height of x's tree. Initially every rank is 0. In `Union`, **make the root of smaller rank point at the root of larger rank**; if the ranks are equal, pick either, and increase the winner's rank by 1.

Why it works. The depth of the combined tree is the depth of the deeper of the two, unless the two are equally deep, in which case it grows by one. Union by rank makes the shallower tree hang off the deeper one, so the deeper one's depth does not change at all. Depth can only grow when two trees of the *same* rank meet.

The bound this gives. A tree of rank r contains at least 2^r nodes. Proof by induction on r. *Base:* a tree of rank 0 is a single node, and $2^0 = 1$. *Step:* a tree of rank r is created only when two trees of rank $r - 1$ are united; by the inductive hypothesis each has at least 2^{r-1} nodes, so the union has at least $2^{r-1} + 2^{r-1} = 2^r$ nodes. ■

Since no tree can contain more than n nodes, $2^r \leq n$, so $r \leq \lg n$. The height is at most the rank, so **every `FindSet` costs at most** $\lg n$, and this is a genuine worst-case bound with no amortization involved. That alone takes Kruskal from $O(E\,V)$ to $O(E \lg V)$.

14.8.3 Trick 2: path compression

Here is the second idea, and it costs almost nothing to implement. When `FindSet(x)` walks from x up to the root r, it passes through every node on the path. On the way back down, **point every one of those nodes directly at r.**

The find that does this pays double — one pass up, one pass down — but every node on that path is now at depth 1, and every future find that touches any of them is instant. The work is not wasted; it is *invested*.

Path compression is why this structure needs amortized analysis at all. A single `FindSet` can still cost $\Theta(\lg n)$ — the first one down a long path does. But it can only do so *once*, because it flattens the path behind it. The cost of the structure is not the worst cost of one operation; it is the total cost of the sequence, and the flattening means later operations inherit the work earlier ones did.

> **WHY THIS IS EXACTLY THIS CHAPTER'S SITUATION.** Compare with the stack. A `Multipop` that removes a thousand items costs a thousand — but those thousand items will never be removed again. A `FindSet` that walks a path of length twenty costs twenty — but those twenty nodes will never be walked again. In both cases the expensive operation is expensive precisely because it is *consuming* structure that earlier operations built, and in both cases the right question is the total over the sequence, not the worst over the operations. The tool for both is the potential method.

14.8.4 What the answer is

Tarjan proved in 1975 [8] that with both tricks, a sequence of m operations on n elements costs

$$O(m\,\alpha(n)),$$

where α is the **inverse Ackermann function**, named after the fast-growing function Ackermann introduced in 1928 [9]. His proof uses a potential function — a sophisticated one, assigning a potential to every node based on its rank and the rank of its parent — and it is several pages long; his monograph [10] develops it at length. We will not reproduce it. What we will do is (a) say honestly what α is, (b) show that it is at most 4 for every n you will ever meet, and (c) measure the real cost of the structure so you can see the theory is not exaggerating.

> **THE THING CHAPTER 11 WARNED ABOUT.** $O(\alpha(n))$ is an **amortized** bound, not a worst-case-per-operation bound. A single `FindSet` genuinely can cost $\Theta(\lg n)$ — the first walk down a long path. What cannot happen is that *many* finds cost that much, because the first one flattens the path. As always: the bound is on the sum.

14.8.5 What α actually is

Most textbooks say "the inverse Ackermann function, which grows extremely slowly" and move on. Here is the honest definition, which is not difficult, only unfamiliar.

Define a family of functions $A_k(j)$, for integers $k \geq 0$ and $j \geq 1$:

$$A_k(j) \;=\; \begin{cases} j+1 & \text{if } k=0, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1, \end{cases} \tag{14.16}$$

where $A_{k-1}^{(j+1)}$ means "apply the function A_{k-1} to its own output, $j+1$ times in a row".

Then

$$\alpha(n) \;=\; \min\{\, k \,:\, A_k(1) \geq n \,\}. \tag{14.17}$$

In words, Equation (14.17) says: $\alpha(n)$ is the smallest level k of that family at which $A_k(1)$ has finally grown as large as n.

Let us compute $A_k(1)$ from Equation (14.16) for the first few k, because the numbers are the whole story.

$A_0(1)$**.** By the first line, $A_0(j) = j+1$, so $A_0(1) = 2$.

$A_1(1)$**.** $A_1(j) = A_0^{(j+1)}(j)$ — apply "add one" $j+1$ times starting from j — which gives $j + (j+1) = 2j+1$. So $A_1(1) = 3$.

$A_2(1)$**.** $A_2(j) = A_1^{(j+1)}(j)$. Since A_1 doubles-and-adds-one, applying it i times to j gives $2^i(j+1) - 1$. With $i = j+1$: $A_2(j) = 2^{j+1}(j+1) - 1$. So $A_2(1) = 2^2 \cdot 2 - 1 = 7$.

$A_3(1)$**.** $A_3(1) = A_2^{(2)}(1) = A_2(A_2(1)) = A_2(7) = 2^8 \cdot 8 - 1 = 2048 - 1 = 2047$.

$A_4(1)$**.** $A_4(1) = A_3^{(2)}(1) = A_3(A_3(1)) = A_3(2047)$, which is A_2 applied to 2047 a total of 2048 times. The very first application already gives $2^{2048} \cdot 2048 - 1$, a number with about 620 digits — and then we do it 2047 more times, each step exponentiating again.

The values of $A_k(1)$ are collected in **Table 14.1**.

Table 14.1 The values $A_k(1)$ that define α. The jump from $k = 3$ to $k = 4$ is not a jump in size, it is a jump in *kind*: 2047 is a number you can write down, and $A_4(1)$ is a number whose decimal expansion 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 the definition of α off **Table 14.1** gives exactly the table Chapter 11 quoted, restated here as **Table 14.2**.

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	$\alpha(n)$
$n \leq 2$	0
$n \leq 3$	1
$n \leq 7$	2
$n \leq 2047$	3
$n \leq A_4(1)$, which exceeds 2^{2048}	4

14.8.6 Why $\alpha(n) \leq 4$ for any n that fits in the universe

The physicists' estimate for the number of atoms in the observable universe is about 10^{80}, which is roughly 2^{266}. Chapter 11 used the same figure.

Now, $A_4(1)$ exceeds 2^{2048}. And 2^{2048} is unimaginably larger than 2^{266} — the ratio is 2^{1782}, itself a number with over five hundred digits.

So: if every atom in the observable universe were a separate element of your disjoint-set structure, n would be about 2^{266}, which is far below $A_4(1)$, and therefore

$$\alpha(n) \;=\; 4.$$

> **WHAT THIS MEANS.** $\alpha(n) \leq 4$ is not "true for realistic inputs" in the hand-waving sense. It is true for every input that could physically be represented, by any machine, ever. So a disjoint-set forest with union by rank and path compression costs **at most four times a constant per operation, amortized** — and in practice far less, because most finds land on a node that has already been compressed. You may treat `FindSet` and `Union` as $O(1)$ in an engineering estimate, while knowing that the honest bound is $O(\alpha(n))$ and that α is not literally constant. Fredman and Saks [11] proved in 1989 that $\Omega(\alpha(n))$ amortized is unavoidable for any pointer-based structure, so nobody is going to improve on this.

14.8.7 Measuring it

Theory this extreme deserves a measurement. **Listing 14.10** builds the forest four ways — with and without each trick — and counts every pointer followed, on two workloads: a chain, which is the worst case for naive union, and a balanced merge, which is the worst case for union by rank.

Listing 14.10 Union–find with both tricks, every pointer counted. Compile with `g++ -std=c++17 -O2 -o dsu dsu.cpp`.

```
// Union-find measured.  Every pointer followed is counted, so the
// "amortized cost" printed below is a measurement, not a quotation.
// Two workloads are used: a chain, which destroys the naive structure,
// and a balanced merge, which is the worst case for union by rank.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

struct Forest {
    vector<int> parent, rnk;
    bool byRank, compress;
    long long hops = 0, ops = 0;           // hops = pointers followed = actual cost

    Forest(int n, bool r, bool c) : parent(n), rnk(n, 0), byRank(r), compress(c) {
        for (int i = 0; i < n; ++i) parent[i] = i;
    }

    int findSet(int x) {
        ++ops;
        int root = x;
        while (parent[root] != root) { root = parent[root]; ++hops; }
        if (compress)                       // second pass: re-point every node
            while (parent[x] != root) { int nxt = parent[x]; parent[x] = root; x = nxt;
++hops; }
        return root;
    }

    void unionSets(int a, int b) {
        int ra = findSet(a), rb = findSet(b);
        if (ra == rb) return;
        if (!byRank) { parent[ra] = rb; return; }        // arbitrary: the bad way
        if (rnk[ra] < rnk[rb]) parent[ra] = rb;
        else if (rnk[ra] > rnk[rb]) parent[rb] = ra;
        else { parent[rb] = ra; ++rnk[ra]; }             // a tie raises the rank by 1
    }

    int height(int x) const { int h = 0; while (parent[x] != x) { x = parent[x]; ++h; }
return h; }
    int tallest() const {
        int t = 0; for (size_t i = 0; i < parent.size(); ++i) t = max(t, height((int)i));
return t;
    }
};
```

Listing 14.10 (Continued) *Union–find with both tricks, every pointer counted. Compile with g++ -std=c++17 -O2 -o dsu dsu.cpp.*

```
// alpha(n), from A_k(1) = 2, 3, 7, 2047, A_4(1) > 2^2048.
int alphaOf(long long n) {
    if (n <= 2) return 0;
    if (n <= 3) return 1;
    if (n <= 7) return 2;
    if (n <= 2047) return 3;
    return 4;                               // true for every n a computer can address
}

void row(const string& name, const Forest& F, int tallBefore, int tallAfter) {
    cout << left << setw(30) << name << right
         << " ops=" << setw(9) << F.ops
         << " hops=" << setw(12) << F.hops
         << " hops/op=" << setw(11) << fixed << setprecision(3)
         << (double)F.hops / (double)F.ops
         << "   height " << tallBefore << " -> " << tallAfter << "\n";
}

// Workload 1: unite 0 with 1, 2, 3, ... in turn.  Each union makes the
// new element the root, so without union by rank a path of length n-1
// grows.  Then find every element once, worst first.
void chain(const string& name, bool r, bool c, int n) {
    Forest F(n, r, c);
    for (int i = 1; i < n; ++i) F.unionSets(0, i);
    int before = F.tallest();
    for (int i = 0; i < n; ++i) F.findSet(i);
    row(name, F, before, F.tallest());
}

// Workload 2: the worst case for union by rank.  Unite equal-rank roots
// in pairs, doubling the tree size each round, which is the only way to
// make the rank -- and so the height -- reach lg n.  Then find every
// element once, deepest first.
void balanced(const string& name, bool r, bool c, int n) {
    Forest F(n, r, c);
    for (int step = 1; step < n; step *= 2)
        for (int i = 0; i + step < n; i += 2 * step) F.unionSets(i, i + step);
    int before = F.tallest();
    vector<int> order(n);
    for (int i = 0; i < n; ++i) order[i] = i;
    sort(order.begin(), order.end(),
         [&F](int a, int b) { return F.height(a) > F.height(b); });
    for (int x : order) F.findSet(x);
    row(name, F, before, F.tallest());
}
```

Listing 14.10 (Continued) *Union–find with both tricks, every pointer counted. Compile with g++ -std=c++17 -O2 -o dsu dsu.cpp.*

```
int main() {
    const int n = 32768;
    cout << "n = " << n << " elements.  alpha(" << n << ") = " << alphaOf(n) << ".\n";
    cout << "lg n = 15, so union by rank alone promises height at most 15.\n";

    cout << "\nWORKLOAD 1  unite 0 with 1,2,3,...,n-1, then find every element.\n";
    chain("  naive (no rank, no compress)", false, false, n);
    chain("  union by rank only",           true,  false, n);
    chain("  path compression only",        false, true,  n);
    chain("  rank + path compression",      true,  true,  n);

    cout << "\nWORKLOAD 2  unite equal-rank roots in pairs, then find every element.\n";
    balanced("  naive (no rank, no compress)", false, false, n);
    balanced("  union by rank only",           true,  false, n);
    balanced("  path compression only",        false, true,  n);
    balanced("  rank + path compression",      true,  true,  n);

    cout << "\nHow alpha grows, from A_k(1) = 2, 3, 7, 2047, ... :\n"
            "   n <= 2       alpha(n) = 0\n"
            "   n <= 3       alpha(n) = 1\n"
            "   n <= 7       alpha(n) = 2\n"
            "   n <= 2047    alpha(n) = 3\n"
            "   n <= A_4(1)  alpha(n) = 4     and A_4(1) is larger than 2^2048\n";
    return 0;
}
```

```
n = 32768 elements.  alpha(32768) = 4.
lg n = 15, so union by rank alone promises height at most 15.

WORKLOAD 1  unite 0 with 1,2,3,...,n-1, then find every element.
  naive (no rank, no compress) ops=    98302 hops=  1073676289 hops/op=  10922.222   height
32767 -> 32767
  union by rank only           ops=    98302 hops=       32767 hops/op=      0.333   height
1 -> 1
  path compression only        ops=    98302 hops=      196595 hops/op=      2.000   height
32766 -> 1
  rank + path compression      ops=    98302 hops=       32767 hops/op=      0.333   height
1 -> 1

WORKLOAD 2  unite equal-rank roots in pairs, then find every element.
  naive (no rank, no compress) ops=    98302 hops=      311264 hops/op=      3.166   height
15 -> 15
  union by rank only           ops=    98302 hops=      245760 hops/op=      2.500   height
15 -> 15
  path compression only        ops=    98302 hops=      163775 hops/op=      1.666   height
14 -> 1
  rank + path compression      ops=    98302 hops=       98271 hops/op=      1.000   height
15 -> 1

How alpha grows, from A_k(1) = 2, 3, 7, 2047, ... :
   n <= 2       alpha(n) = 0
   n <= 3       alpha(n) = 1
   n <= 7       alpha(n) = 2
   n <= 2047    alpha(n) = 3
   n <= A_4(1)  alpha(n) = 4     and A_4(1) is larger than 2^2048
```

Five things to read out of that.

The naive structure is catastrophic. On workload 1 it averages 10,922 pointer hops per operation. It builds a path of height 32,767 and never flattens it. The measured cost per operation is about $n/3$, which is $\Theta(n)$.

Union by rank alone is enough to stop the catastrophe. On workload 1 it never lets the height exceed 1.

But union by rank alone is not the end of the story. Workload 2 is designed to defeat it: by uniting equal-rank roots in pairs, we force the rank up one level at a time until it reaches 15. And 15 is exactly lg32768 — the bound proved in Section 14.8.2, achieved to the unit. Union by rank does not *promise* better than lg$n$, and here it does not deliver better.

Path compression finishes the job. On workload 2, "rank + path compression" flattens the height-15 forest down to height 1 and averages exactly 1.000 hops per operation, against 2.500 for union by rank alone.

The measurements are nowhere near $\alpha(n) = 4$. The proved bound is $O(m\,\alpha(n))$ with $\alpha(32768) = 4$; the measured cost is 1.000 hops per operation. The bound is an upper bound with room to spare, and in ordinary use the structure behaves as if it were flatly constant-time — which, for engineering purposes, it is.

That is the debt from Chapter 11 discharged. `FindSet` and `Union` cost $O(\alpha(n))$ amortized, $\alpha(n) \leq 4$ always, and the reason it is an amortized bound rather than a worst-case one is that path compression makes each operation's cost depend on what earlier operations did — which is the situation this entire chapter is about.

14.9 Choosing a Method

Three methods, one answer. Which should you reach for? **Table 14.3** sets them side by side.

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
Weakest when	the operations differ a lot in cost	there is no obvious object to attach credit to	you cannot guess Φ
Examples in this chapter	stack §14.3.2, counter §14.3.3, Exercise 17.1-3 §14.3.4	stack §14.5.3, counter §14.5.4, Exercise 17.2-2 §14.5.5, table §14.7.4	stack §14.6.4, counter §14.6.5, queue-from-stacks §14.6.6, tables §14.7.5 and §14.7.7
Scales to	simple structures only	moderately complex structures	splay trees, Fibonacci heaps, union–find

A practical order of attack:

1. **Try the aggregate method first.** It is the shortest, and when a clean counting argument exists it is the most convincing. If you can say "each element is touched at most twice over the whole sequence", stop — you are done.

2. **Reach for the accounting method when the operations have genuinely different costs** and you want to say so. It is also the easiest to explain to someone who has not studied the subject, because the metaphor is honest bookkeeping.
3. **Reach for the potential method when the other two fail**, or when the structure's expensive event depends on its *shape* rather than on any particular object — a table about to double, a tree about to rebalance, a heap about to consolidate. It is also the method to learn properly, because it is the only one of the three that scales to the hard structures.

> **THEY ARE NOT DIFFERENT THEOREMS.** All three prove statements of the same form: $\sum c_i \leq \sum \hat{c}_i$. Section 14.6.3 showed that the potential method's condition and the accounting method's credit invariant are literally the same inequality. So if you get different answers from two methods, one of them contains an arithmetic error — not a deeper truth.

14.10 Profitina in Practice

14.10.1 Where amortization lives inside an AI-visibility platform

Profitina is an AI Visibility (GEO — Generative Engine Optimization) platform: it scans a brand's website, scores how ready that site is to be **cited by an AI answer engine**, explains each signal it measured, and then helps fix what it found. It runs in production at `app.profitina.com`, on a Python web application over PostgreSQL, with local model inference through Ollama.

Three parts of that system are amortized-analysis problems in the strict sense of this chapter.

The crawl frontier is a dynamic table. When the scanner starts on a site it does not know how many pages it will find. It discovers links as it goes, and every discovered URL is appended to a frontier. That frontier is a growing array, and it is precisely the structure of Section 14.7: append is $O(1)$ amortized, occasionally $\Theta(m)$ when the block doubles. The engineering consequence is exact and practical: **the scanner's total memory-copy work over a crawl of m pages is $O(m)$, not $O(m^2)$**, and a page that triggers a doubling is not a bug report. Without the amortized argument, a crawl of 50,000 pages looks like it might do 2.5×10^9 units of copying; with it, we know it does fewer than 1.5×10^5.

The signal buffer is a two-stack queue. Signals — content extractability, structured-data presence, technical-authority markers — are produced by the crawler in whatever order pages finish, and consumed by the scorer in a fixed order. The buffer between them must be FIFO, and it is built from two stacks exactly as in Section 14.6.6, so that a producer never blocks: `Enqueue` is one push, amortized 3. When the consumer drains, one `Dequeue` does $\Theta(e)$ work and the rest do $O(1)$, and the *total* is linear. The dashboard's latency budget is written against the amortized figure, not the worst single figure — which is legitimate for throughput, and Section 14.11 explains when it would not be.

The re-scan scheduler is a counter problem in disguise. Each tracked brand is re-scanned at a cadence that depends on its subscription tier, and the scheduler must decide, every hour, which brands are due. A naive scheduler examines every brand every hour, at cost $\Theta(B)$ per tick for B brands. A better one maintains a bucketed structure where each brand appears in exactly one bucket and buckets are promoted the way a binary counter's bits are: most ticks touch one bucket, and a rare tick cascades through several. The analysis is the k-bit counter of Section 14.3.3 with different nouns, and it gives **amortized $O(1)$ scheduling work per tick regardless of how many tiers exist**.

Near-duplicate detection reuses union–find. Chapter 11 noted that Profitina groups near-identical pages before scoring, so that a site with a thousand auto-generated pages is not scored a thousand times for the same content. Those groups are disjoint sets, they merge as similarities are discovered, and the structure is the forest of Section 14.8. The $O(\alpha(n))$ bound is what allows the deduplication pass to be costed as linear in the number of similarity pairs.

14.10.2 The worked case study: Profitina Laundry

The hands-on case study in this book is always **Profitina Laundry**, a small teaching business, not the real product.

Profitina Laundry takes in garments at a counter. Three structures from this chapter do three jobs.

1. **The intake register** is a dynamic table. Every garment that arrives is logged in it. It doubles when full.
2. **The pending pile** is a stack. Garments wait there for a machine. Loading a machine takes up to twelve garments at once — a `Multipop`.
3. **The collection line** is a queue built from two stacks, so that customers collect in the order they arrived even though the underlying piles are last-in-first-out.

The manager's question is not academic. She wants to know whether the counter software will keep up **on the worst day** — the day a hotel sends its entire linen store at eight in the morning and nothing is washed until five in the afternoon. **Listing 14.11** simulates an ordinary Tuesday and that worst day, counting every unit of work.

Listing 14.11 Profitina Laundry: a dynamic table, a `Multipop` stack and a two-stack queue, all instrumented, over one working day. Compile with `g++ -std=c++17 -O2 -o laundry laundry.cpp`.

```
// Profitina Laundry -- one working day at the intake counter, with every
// structure in this chapter doing a job and every unit of work counted.
//
//   * the INTAKE REGISTER is a dynamic table that doubles when full;
//   * the PENDING PILE is a stack, and loading a machine is a MULTIPOP;
//   * the COLLECTION LINE is a FIFO queue built from two stacks.
//
// The program runs an ordinary day and then the worst day the manager
// can imagine, and reports the measured amortized cost per event.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

struct Register {                       // the dynamic table
    vector<string> slot;
    long long size = 0, work = 0, expansions = 0;
    void add(const string& tag) {
        if (size == 0) { size = 1; slot.reserve(1); }
        if ((long long)slot.size() == size) {
            work += size;               // copy every existing entry
            size *= 2; ++expansions;
        }
        slot.push_back(tag); work += 1;
    }
    long long potential() const { return 2 * (long long)slot.size() - size; }
};

struct Pile {                           // the stack, with MULTIPOP
    vector<string> item;
    long long work = 0;
    void put(const string& t) { item.push_back(t); work += 1; }
    int load(int capacity) {            // MULTIPOP: fill one machine
        int taken = 0;
        while (!item.empty() && taken < capacity) { item.pop_back(); ++work; ++taken; }
        return taken;
    }
    long long potential() const { return (long long)item.size(); }
};
```

Listing 14.11 (Continued) *Profitina Laundry: a dynamic table, a Multipop stack and a two-stack queue, all instrumented, over one working day. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
struct Line {                           // the queue from two stacks
    vector<int> in, out;
    long long work = 0;
    void join(int ticket) { in.push_back(ticket); work += 1; }
    int serve(bool& ok) {
        if (out.empty())
            while (!in.empty()) { out.push_back(in.back()); in.pop_back(); work += 2; }
        if (out.empty()) { ok = false; return -1; }
        int t = out.back(); out.pop_back(); work += 1; ok = true; return t;
    }
    long long potential() const { return 2 * (long long)in.size(); }
};

struct Day {
    Register reg; Pile pile; Line line;
    long long events = 0;
    long long work() const { return reg.work + pile.work + line.work; }
    long long potential() const { return reg.potential() + pile.potential() +
line.potential(); }
};

void report(const string& name, const Day& d, int garments) {
    cout << left << setw(26) << name << right
         << " garments=" << setw(7) << garments
         << " events=" << setw(8) << d.events
         << " work=" << setw(9) << d.work()
         << " work/event=" << setw(9) << fixed << setprecision(4)
         << (double)d.work() / (double)d.events
         << "  expansions=" << d.reg.expansions << "\n";
}

int main() {
    const int MACHINE = 12;             // one machine holds 12 garments

    // ---- An ordinary Tuesday: garments trickle in, a machine is loaded
    // ---- whenever twelve are waiting, customers collect in order.
    {
        Day d; int garments = 20000; int ticket = 0; bool ok;
        for (int g = 1; g <= garments; ++g) {
            d.reg.add("G" + to_string(g)); ++d.events;
            d.pile.put("G" + to_string(g)); ++d.events;
            if ((int)d.pile.item.size() >= MACHINE) {
                int n = d.pile.load(MACHINE); ++d.events;
                for (int i = 0; i < n; ++i) { d.line.join(++ticket); ++d.events; }
            }
            if (g % 3 == 0) { d.line.serve(ok); ++d.events; }
        }
```

Listing 14.11 (Continued) *Profitina Laundry: a dynamic table, a Multipop stack and a two-stack queue, all instrumented, over one working day. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
        report("ordinary Tuesday", d, garments);
    }

    // ---- The worst day: a hotel sends its whole linen store at 08:00,
    // ---- nothing is washed until 17:00, then everything at once.
    {
        Day d; int garments = 20000; int ticket = 0; bool ok;
        for (int g = 1; g <= garments; ++g) {
            d.reg.add("H" + to_string(g)); ++d.events;
            d.pile.put("H" + to_string(g)); ++d.events;
        }
        while (!d.pile.item.empty()) {                  // one huge MULTIPOP run
            int n = d.pile.load(garments); ++d.events; // capacity = everything
            for (int i = 0; i < n; ++i) { d.line.join(++ticket); ++d.events; }
        }
        for (int i = 0; i < garments; ++i) { d.line.serve(ok); ++d.events; }
        report("worst day (one big load)", d, garments);
    }

    // ---- What a single event can cost on the worst day, and what the
    // ---- whole day costs per event.
    {
        Day d; int garments = 20000;
        long long worst = 0;
        for (int g = 1; g <= garments; ++g) {
            long long before = d.reg.work;
            d.reg.add("W" + to_string(g)); ++d.events;
            worst = max(worst, d.reg.work - before);
        }
        cout << "\nIntake register alone: " << garments << " garments logged.\n";
        cout << "  most expensive single log entry = " << worst
             << " slot writes (a table expansion)\n";
        cout << "  total work                      = " << d.reg.work << "\n";
        cout << "  amortized cost per garment      = " << fixed << setprecision(4)
             << (double)d.reg.work / (double)garments << " slot writes\n";
        cout << "  aggregate bound proved in 14.7  = 3 per garment, i.e. "
             << 3 * garments << " in total\n";
        cout << "  final potential 2*num - size    = " << d.reg.potential() << "\n";
    }
    return 0;
}
```

```
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
```

Read that as the manager would.

The worst day costs **2.16 units of work per event**. The ordinary Tuesday costs **2.13**. The difference is under two percent. The catastrophe she was worried about — twenty thousand garments arriving at once and nothing washed until the evening — does not cost the software meaningfully more than a quiet day, because every structure involved has an amortized bound that does not care about the *order* of the events, only their number.

And look at the single worst event: **16,385 slot writes**, from one table expansion. That one log entry is sixteen thousand times more expensive than an ordinary one. If the counter software had a hard deadline of, say, one millisecond per garment, that single entry might miss it. It does not, because the work is a memory copy of 16,385 pointers, which takes microseconds — but the *shape* of the risk is real, and Section 14.11 says what to do about it.

Finally, compare the measured 2.64 writes per garment against the proved bound of 3. The theory is correct and not far from tight.

> **WHAT THE MANAGER SHOULD TAKE AWAY.** She does not need to buy a bigger computer for busy days. She needs to know that the software's cost grows **linearly with the number of garments**, and that peaks in arrival rate do not change that. If someone offers her a "faster" intake system that grows its register by a fixed hundred slots at a time, she should refuse: by **Listing 14.13** that system costs $\Theta(n)$ per garment, and on a twenty-thousand-garment day its register alone would do $20{,}000 + 100 \cdot \frac{200 \cdot 199}{2} = 2{,}010{,}000$ slot writes — almost forty times the 52,767 of the one she has.

14.11 Common Mistakes, and How to Avoid Them

1. Saying "amortized" when you mean "average-case". The single most common error, and the most damaging. Amortized analysis involves no probability. Its bound holds for every sequence, including the worst. *Cure:* before writing "amortized", ask yourself — where is the probability distribution? If there is one, the word you want is "expected".

2. Saying "amortized" when you mean "usually fast". A structure with amortized cost $O(1)$ might still have one operation in the sequence that costs $\Theta(n)$. The bound is on the total, not on any individual operation. *Cure:* if you need a guarantee about a *single* operation — a real-time deadline, an interactive response, an interrupt handler — an amortized bound is not the guarantee you need. Look for a *worst-case* bound, or use an incremental technique that spreads the expensive work over many operations instead of doing it all at once.

3. Forgetting that amortized bounds do not compose with nested loops the way you hope. If a structure gives you amortized $O(1)$ per operation and you perform n operations, the total is $O(n)$ — that is fine. But if you perform n operations *inside* an outer loop that runs n times, **and the structure is reset between outer iterations**, you get $n \times O(n) = O(n^2)$, which may be right. The amortization only applies within one uninterrupted sequence on one structure. *Cure:* identify exactly which sequence the bound is about.

4. Choosing a potential function and forgetting to check $\Phi(D_0)$. Every potential-method proof has two obligations: show $\Phi(D_i) \geq \Phi(D_0)$ for all i, as Equation (14.12) demands, and compute $\hat{c}_i$. Beginners do the second and skip the first. Without the first, the proof is worthless — you can "prove" any bound at all with a potential function that is allowed to go arbitrarily negative. *Cure:* write the two lines "$\Phi(D_0) = 0$" and "$\Phi(D) \geq 0$ for every reachable state, because it counts something" before you compute a single amortized cost.

5. Asserting that the credit stays non-negative instead of proving it. The accounting method is only a proof when you say *where the credit sits*: on this item, on that 1 bit, on those newly inserted table entries. Then non-negativity is obvious because you are counting objects. *Cure:* state the credit rule as an invariant, in one sentence, and check it against each operation separately.

6. Halving a dynamic table at half-full. The trap of Section 14.7.6, and it is a real bug that has been shipped in real libraries. The expansion and contraction thresholds must be separated. *Cure:* expand at load factor 1, contract at load factor 1/4, so that the structure sits at 1/2 after either event and must be pushed a constant fraction of its size to trigger the next one.

7. Growing a table by a constant instead of a constant factor. `size += 100` looks thriftier than `size *= 2` and is asymptotically catastrophic: $\Theta(n^2)$ total copying instead of $\Theta(n)$. *Cure:* always grow by a *factor*. The factor need not be 2 — 1.5 is common, and Exercise 14.9 shows that any constant factor greater than 1 works — but it must be a factor.

8. Treating the `Multipop` argument as if it needed the stack to start empty. It does not need it to be empty; it needs it to be a *known* starting state whose potential you can compute. If the stack starts with s_0 items, the potential starts at s_0 rather than 0, and the total becomes $2n + s_0$. *Cure:* when the structure does not start empty, add $\Phi(D_0)$ to the total and say so.

9. Confusing the two α's. The load factor of a dynamic table and the inverse Ackermann function are both written α. They have nothing to do with each other. *Cure:* read the sentence around it. If it is a fraction between 0 and 1, it is a load factor.

10. Believing that $O(\alpha(n))$ means $O(1)$. It does not, formally — α is unbounded, it just grows so slowly that its value is 4 for every input a machine can hold. Writing $O(1)$ in a proof is wrong; writing "effectively constant" in an engineering note is fine. *Cure:* keep the honest bound in the analysis and the practical reading in the commentary, as Section 14.8.6 does.

14.12 Chapter Summary

The question. Given a sequence of n operations on a data structure, what does the *whole sequence* cost? Multiplying the worst single operation by n can overstate the answer by a factor of n, because expensive operations are paid for by the cheap ones that made them possible.

The definition. An assignment of amortized costs $a_1, \ldots, a_n$ is valid when $\sum_{i \le k} c_i \le \sum_{i \le k} a_i$ for every prefix k. **Figure 14.1** states the requirement and — in red — the thing that is *not* required: $c_i \le a_i$ for individual i.

Not average-case. Twice stated, because it is the misconception beginners carry: there is no probability in amortized analysis, and the bound holds for every sequence, including an adversary's.

The aggregate method. Compute $T(n)$ directly and divide. On the stack with `Multipop`, "each element is popped at most once after it is pushed" gives $T(n) \le 2n$. On the k-bit counter, bit i flips $\lfloor n/2^i \rfloor$ times and $\sum_{i \ge 0} n/2^i = 2n$. On Exercise 17.1-3, splitting the sum into a baseline of n and a geometric series of extras gives the exact total $2^{\lfloor \lg n \rfloor + 1} + n - \lfloor \lg n \rfloor - 2$, hence $\le 3n - \lfloor \lg n \rfloor - 2$.

The accounting method. Charge each operation type a fixed amount and store the surplus as credit on specific objects; prove the credit never goes negative by identifying what it counts. Stack: charge 2 for `Push`, 0 for `Pop` and `Multipop`, credit = number of items. Counter: charge 2 per bit set to 1, credit = number of 1 bits. Exercise 17.2-2: charge 3 per operation, credit $= 2j + k + 2$ where $i = 2^k + j$.

The potential method. Define Φ on the structure's state and set $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$. Summing telescopes to $\sum \hat{c}_i = \sum c_i + \Phi(D_n) - \Phi(D_0)$, so the method is valid provided $\Phi(D_i) \ge \Phi(D_0)$. Stack: Φ = number of items. Counter: Φ = number of 1 bits. Two-stack queue: $\Phi = 2 \times |S_{IN}|$, giving `Enqueue` 3 and `Dequeue` 1.

Dynamic tables. `TABLE-INSERT` with doubling: expansions happen when $i - 1$ is a power of 2 and cost i; the total is under $3n$; the accounting charge is 3 per item (itself, its own relocation, and one older item's relocation); the potential is $\Phi = 2\,num - size$, which gives exactly 3 in both cases and falls to 0 at every expansion, as **Figure 14.19** shows. With deletion, halving at half-full thrashes — the block `INSERT, DELETE, DELETE, INSERT` costs $\Theta(size)$ every four operations, measured at over 32,000 units per operation at $n = 262{,}144$ — while halving at quarter-full costs 1.50 per operation at every scale and keeps the load factor in $[1/4,1]$.

Union–find. Union by rank alone bounds the height by $\lg n$, proved by "a tree of rank r has at least 2^r nodes". Path compression makes the analysis amortized, because a costly find flattens the path behind it. Together they give $O(m\,\alpha(n))$, with α defined from the Ackermann family $A_k(1) = 2,3,7,2047,\ldots$, so that $\alpha(n) \le 4$ for every n below $A_4(1) > 2^{2048}$ — every n that could physically exist.

Choosing. Aggregate when a clean counting argument exists; accounting when credit attaches to objects; potential when the trouble is in the shape of the structure — and potential when the structure is hard, because it is the only one of the three that scales.

14.12.1 Looking ahead

Chapter 15 returns to shortest paths and asks for all pairs at once, where the technique is matrix-like repeated relaxation rather than amortization. But the habit this chapter builds — *ask what the whole sequence costs, not what the worst step costs* — is one you will now use everywhere. Every time you write a loop that calls `push_back`, every time you use a hash table that rehashes, every time you merge a Fibonacci heap, you are relying on an amortized bound. You now know what those bounds mean, how they are proved, and — just as importantly — what they do not promise.

14.13 Exercises

14.1 A stack, initially empty, receives n `Push` operations followed by a single `Multipop(S, n)`. How many operations are in the sequence, what is the total actual cost, and what is the average cost per operation? Which of the three methods is the natural one to use here, and why?

14.2 Write out, as a formal injection, the argument of Section 14.3.2 that the number of removals from a stack never exceeds the number of pushes. State clearly what the function maps, and prove that it is well defined and one-to-one.

14.3 A k-bit binary counter does **not** start at zero: it starts holding some value whose binary representation contains b_0 ones. Using the potential method, give the best bound you can on the total cost of n `Increment` operations from that state. Under what condition on b_0 is the total still $O(n)$?

14.4 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 **3**, and 1 otherwise. Use the aggregate method to find the amortized cost per operation, giving the constant explicitly.

14.5 Add an operation `Multipush(S, k)` to the stack, which pushes k copies of some item. Is the amortized cost of the three operations `Push`, `Multipop` and `Multipush` still $O(1)$? Justify your answer with an explicit sequence, and say what goes wrong with the potential function Φ = number of items.

14.6 The potential method as stated assumes $\Phi(D_0) = 0$. Suppose it is not. Derive the correct relationship between $\sum c_i$ and $\sum \hat{c}_i$, and state precisely what must be assumed for the amortized costs still to bound the actual ones.

14.7 Design a k-bit counter that supports `Increment` and also `Reset`, which returns the counter to zero. Both operations must be $O(1)$ amortized. Give the data, the two procedures in outline, a potential function, and the amortized cost of each operation. (A `Reset` that simply writes k zeros will not do: show why.)

14.8 Show that the aggregate method is a special case of the potential method, by exhibiting a potential function which, for any structure with aggregate bound $T(n)$, assigns every operation the amortized cost $T(n)/n$. What extra condition must the sequence satisfy for your Φ to be legitimate?

14.9 A dynamic table grows by a factor of 3 rather than 2: when it becomes full, a new table with $3 \cdot size$ slots is allocated. Find a potential function of the form $\Phi(T) = a \cdot num[T] - b \cdot size[T]$ that works, and compute the amortized cost of `TABLE-INSERT`. Then generalize to a growth factor $\gamma > 1$.

14.10 A dynamic table grows by adding a fixed 100 slots each time it fills, instead of multiplying its size. Show that the amortized cost of `TABLE-INSERT` is $\Theta(n)$, and give the exact total cost of n insertions.

14.11 Suppose we contract the table when the load factor drops below $1/3$ rather than $1/4$. Does the amortized bound survive? Give a potential function for the combined case and justify it, or give a sequence that breaks it.

14.12 Complete the case analysis for the combined dynamic table of Section 14.7.7. Show that with the stated potential function, `TABLE-INSERT` has amortized cost at most 3 and `TABLE-DELETE` at most 2, in every case.

14.13 In the two-stack queue of Exercise 17.3-6, suppose that when S_{OUT} is empty we move only *half* the items from S_{IN}, leaving the rest behind. What goes wrong? Be precise: is the problem the cost, the correctness, or both?

14.14 Prove by induction that in a disjoint-set forest maintained with union by rank, a tree whose root has rank r contains at least 2^r nodes. Deduce that no tree has height greater than $\lfloor \lg n \rfloor$.

14.15 Compute $A_3(1)$ exactly from the definition in Section 14.8.5, showing each application of A_2. Then show that $A_4(1) > 2^{2048}$, and conclude that $\alpha(n) \leq 4$ for every n less than 2^{2048}.

14.16 Write a program that measures what `std::vector<int>::push_back` actually does: record the capacity after every push, report every reallocation and the growth ratio, and count the total number of element copies over a million pushes. Compare the measurement with the $3n$ bound of Section 14.7.3.

14.17 A k-bit counter supports `Increment` and `Decrement`. Show that the amortized cost per operation is **not** $O(1)$, by exhibiting a sequence of n operations costing $\Theta(nk)$. Explain why the potential function Φ = number of 1 bits fails.

14.14 Solutions to the Exercises

14.1 There are $n + 1$ operations: n pushes and one `Multipop`.

The n pushes cost 1 each, for n. The `Multipop(S, n)` finds n items on the stack and removes all of them, costing n. So the total actual cost is

$$T = n + n = 2n.$$

The average per operation is $2n/(n+1) < 2$.

The natural method is the **aggregate method**, because the whole sequence is given explicitly and its total can be computed directly; there is nothing to distribute. Note that the answer, <2, is exactly the constant that the aggregate analysis of Section 14.3.2 predicted for an arbitrary sequence — this sequence is the one that attains it.

14.2 Let R be the set of removal events in the sequence (each removal of one item from the stack, whether it happened inside a `Pop` or inside a `Multipop` loop), and let P be the set of `Push` operations. Define

$$f : R \to P$$

by: $f(\rho)$ is the `Push` operation that placed on the stack the item that removal ρ removes.

f is well defined. Every item on the stack got there by exactly one `Push`; there is no other way onto the stack. So for each removal event the item it removes has a unique originating push.

f is one-to-one. Suppose $f(\rho_1) = f(\rho_2) = \pi$ for removals $\rho_1 \neq \rho_2$. Then ρ_1 and ρ_2 both remove the item placed by π. But once that item is removed it is no longer on the stack, and nothing returns it — a later `Push` of an equal value is a *different* push, placing a *different* occurrence. So whichever of ρ_1, ρ_2 comes second removes an item that is not there, which is impossible. Hence $\rho_1 = \rho_2$, a contradiction.

An injection from R into P gives $|R| \leq |P|$, which is the claim. Since $|P| \leq n$, we get $|R| \leq n$, and the total cost $|P| + |R| \leq 2n$. ■

14.3 Keep the potential $\Phi(D_i) = b_i$, the number of 1 bits after operation i. The analysis of Section 14.6.5 is unchanged: every increment still has $\hat{c}_i \leq 2$, because that derivation used only $c_i \leq t_i + 1$ and $b_i \leq b_{i-1} - t_i + 1$, neither of which mentions the starting value.

What changes is the accounting at the ends. From the telescoping identity Equation (14.11),

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0) \leq 2n - b_n + b_0 \leq 2n + b_0,$$

using $b_n \geq 0$. So the total cost is at most $2n + b_0$.

Since the counter has only k bits, $b_0 \leq k$, giving the safe bound $2n + k$.

The total is $O(n)$ provided $b_0 = O(n)$ — in particular whenever $k = O(n)$, which is the case in any realistic setting. Only if the counter were absurdly wide compared with the number of operations (say $k = n^2$) would the starting configuration dominate, and even then the cost is $O(n + k)$, which is the honest answer: **you may have to pay once for the state you were handed, but only once.**

14.4 Write c_i for the cost of operation i, so $c_i = i$ when $i = 3^j$ and $c_i = 1$ otherwise.

Split as in Section 14.3.4. Every operation pays a baseline of 1, contributing n. The operations at $i = 3^j$ pay an extra $3^j - 1$. The powers of 3 not exceeding n are those with $j \le \lfloor \log_3 n \rfloor$; write $m = \lfloor \log_3 n \rfloor$.

$$\text{Total} \;=\; n + \sum_{j=0}^{m} (3^j - 1) \;=\; n + \left(\sum_{j=0}^{m} 3^j \right) - (m+1).$$

By the geometric-series formula with $x = 3$,

$$\sum_{j=0}^{m} 3^j \;=\; \frac{3^{m+1} - 1}{3 - 1} \;=\; \frac{3^{m+1} - 1}{2}.$$

Since $3^m \le n$, we have $3^{m+1} \le 3n$, so this is at most $(3n - 1)/2 < 3/2\, n$.

Therefore

$$\text{Total} \;<\; n + 3/2\, n \;=\; 5/2\, n \;=\; O(n),$$

and the amortized cost per operation is less than $5/2 = 2.5$, so $O(1)$.

The general pattern. With base β in place of 2 or 3, the geometric sum is $(\beta^{m+1} - 1)/(\beta - 1) \le \beta n/(\beta - 1)$, so the amortized cost is at most $1 + \beta/(\beta - 1)$. For $\beta = 2$ that is 3, matching Section 14.3.4; for $\beta = 3$ it is 2.5; and as β grows it falls toward 2. **The rarer the expensive operations, the smaller the constant — but it never drops below 2.**

14.5 No. The amortized cost is not $O(1)$; it is $\Theta(n)$ in the worst case.

The sequence. Perform $n/2$ operations, each `Multipush(S, n/2)`, then $n/2$ operations, each `Multipop(S, 1)`. Each `Multipush` genuinely pushes $n/2$ items, so its actual cost is $n/2$. The total cost of the first half is

$$\frac{n}{2} \times \frac{n}{2} \;=\; \frac{n^2}{4},$$

and the second half adds only $n/2$. So n operations cost $\Theta(n^2)$, and the amortized cost per operation is $\Theta(n)$.

Why Φ = number of items fails. With that potential, `Multipush(S,k)` has

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}) \;=\; k + k \;=\; 2k,$$

which is not bounded by a constant — it grows with k. The potential *doubles* the charge instead of canceling it.

The reason, in words. `Multipop` works because it **consumes** stored potential: it does k' units of work and destroys k' units of stored obligation, so the two cancel. `Multipush` does the opposite: it does k units of work **and** creates k units of stored obligation. There is nothing to cancel against, and no accounting trick can hide work that is genuinely being performed for the first time.

> **THE GENERAL RULE THIS ILLUSTRATES.** Amortization can only make an operation look cheap if that operation is *destroying* structure that earlier operations paid to build. It can never make an operation that *builds* k units of structure cost less than k. If someone claims an amortized $O(1)$ bound for an operation that creates unboundedly much new state, they are wrong.

14.6 Start from the telescoping identity Equation (14.11) of Section 14.6.2, which holds whatever $\Phi(D_0)$ is:

$$\sum_{i=1}^{n} \hat{c}_i \;=\; \sum_{i=1}^{n} c_i \;+\; \Phi(D_n) - \Phi(D_0).$$

Rearranged:

$$\sum_{i=1}^{n} c_i \;=\; \sum_{i=1}^{n} \hat{c}_i \;-\; \Phi(D_n) \;+\; \Phi(D_0).$$

For the amortized total to bound the actual total we need $-\Phi(D_n) + \Phi(D_0) \leq 0$, i.e. $\Phi(D_n) \geq \Phi(D_0)$. **That is the only assumption required** — and it must hold for every prefix, i.e. $\Phi(D_k) \geq \Phi(D_0)$ for all k, if the bound is to hold for every prefix.

If instead we only know $\Phi \geq 0$ everywhere, we get the weaker but still useful

$$\sum_{i=1}^{n} c_i \;\leq\; \sum_{i=1}^{n} \hat{c}_i \;+\; \Phi(D_0),$$

which says: **the amortized total bounds the actual total, up to a one-off surcharge equal to the initial potential.** Exercise 14.3 is exactly this situation with $\Phi(D_0) = b_0$.

The usual practice — insisting $\Phi(D_0) = 0$ and $\Phi \geq 0$ — is just the convenient special case in which the surcharge vanishes.

14.7 The design. Keep the bit array $A[0 \ldots k-1]$ as before, plus one extra integer $high$, which records **one past the highest index that has ever been set to 1 since the last reset**. Initially $high = 0$.

- `Increment`: exactly as before — reset the trailing 1s, set the first 0 — and additionally, if the bit set was at index i, update $high \leftarrow \max(high,\, i+1)$.
- `Reset`: write 0 to $A[0], A[1], \ldots, A[high-1]$ only, then set $high \leftarrow 0$. Every bit at index $\geq high$ is already 0 and need not be touched.

Why a naive `Reset` will not do. If `Reset` wrote all k zeros, an adversary could alternate `Increment` and `Reset`: each pair costs $1 + k$, so n operations cost $\Theta(nk)$, and the amortized cost is $\Theta(k)$, not $O(1)$.

The potential.

$$\Phi(D) \;=\; (\text{number of 1 bits in } A) \;+\; high.$$

Requirements. Initially the counter is zero and $high = 0$, so $\Phi(D_0) = 0$. Both terms are counts, so $\Phi \geq 0$ always. Good.

Amortized cost of `Increment`. Suppose it resets t bits and sets the bit at index i. Actual cost $c = t + 1$. The number of 1 bits changes by $1 - t$. And $high$ increases by at most 1 — it increases only if $i + 1 > high$, and since the bit at i was 0 and all bits below were 1, we had $high \geq i$, so the increase is at most 1. Hence

$$\hat{c} \;\leq\; (t + 1) + (1 - t) + 1 \;=\; 3.$$

Amortized cost of `Reset`. Actual cost $c = high$ (we write $high$ zeros; if $high = 0$ charge 1 for the call itself). The number of 1 bits drops to 0, a fall of at least 0; and $high$ drops from its value to 0, a fall of exactly $high$. So

$$\hat{c} \;=\; high + (0 - (\text{ones} + high)) \;=\; -\text{ones} \;\leq\; 0.$$

(In the trivial case $high = 0$ the counter is already zero, nothing changes, and the amortized cost is the 1 charged for the call.) So `Reset` costs at most 1, amortized — its real cost was prepaid by the increments that raised $high$ in the first place.

Both operations are $O(1)$ amortized, so n operations cost $O(n)$. **Listing 14.12** implements and measures it.

Listing 14.12 A counter with `Increment` and `Reset`, both $O(1)$ amortized. Compile with `g++ -std=c++17 -O2 -o resetcounter resetcounter.cpp`.

```
// Solution to Exercise 14.7.  A k-bit counter supporting INCREMENT and
// also RESET, which sets the counter back to zero.  The trick is to
// remember the highest index ever set, so RESET need only clear the bits
// below it.  With the potential Phi = (number of 1 bits) + high, both
// operations are O(1) amortized.  The program measures it.
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

struct ResetCounter {
    vector<int> A;
    int high = 0;                       // one past the highest index ever set to 1
    long long actual = 0, ops = 0, worstOne = 0, resetCost = 0;

    explicit ResetCounter(int k) : A(k, 0) {}

    long long phi() const {
        long long ones = 0; for (int b : A) ones += b;
        return ones + high;             // one unit per 1 bit, one per live index
    }

    void increment() {
        long long c = 0; size_t i = 0;
        while (i < A.size() && A[i] == 1) { A[i] = 0; ++c; ++i; }
        if (i < A.size()) { A[i] = 1; ++c; if ((int)i + 1 > high) high = (int)i + 1; }
        charge(c);
    }

    void reset() {
        long long c = 0;
        for (int i = 0; i < high; ++i) { A[i] = 0; ++c; }   // only the live prefix
        high = 0;
        if (c == 0) c = 1;              // an empty RESET still costs one step
        resetCost += c;
        charge(c);
    }

private:
    void charge(long long c) { actual += c; ++ops; if (c > worstOne) worstOne = c; }
};
```

Listing 14.12 (Continued) *A counter with Increment and Reset, both O(1) amortized. Compile with g++ -std=c++17 -O2 -o resetcounter resetcounter.cpp.*

```
int main() {
    ResetCounter C(32);
    long long lowestPhi = C.phi();
    // The adversary: climb to a large value, RESET, climb again, ...
    for (int round = 0; round < 2000; ++round) {
        for (int t = 0; t < 500; ++t) C.increment();
        C.reset();
        if (C.phi() < lowestPhi) lowestPhi = C.phi();
    }
    cout << "2000 rounds of (500 INCREMENTs then one RESET)\n";
    cout << "  operations               = " << C.ops << "\n";
    cout << "  total actual cost        = " << C.actual << "\n";
    cout << "  most expensive single op = " << C.worstOne << "\n";
    cout << "  measured cost per op     = " << fixed << setprecision(4)
         << (double)C.actual / (double)C.ops << "\n";
    cout << "  lowest potential seen    = " << lowestPhi
         << "  (it must never go below Phi(D_0) = 0)\n";
    cout << "  final potential          = " << C.phi() << "\n";

    cout << "\n  cost of the 2000 RESETs alone = " << C.resetCost << "\n";
    cout << "  a RESET that blindly cleared all 32 bits would have cost "
         << 2000 * 32 << "\n";
    return 0;
}
```

```
2000 rounds of (500 INCREMENTs then one RESET)
  operations               = 1002000
  total actual cost        = 2006000
  most expensive single op = 9
  measured cost per op     = 2.0020
  lowest potential seen    = 0  (it must never go below Phi(D_0) = 0)
  final potential          = 0

  cost of the 2000 RESETs alone = 18000
  a RESET that blindly cleared all 32 bits would have cost 64000
```

Two things confirm the analysis. The measured cost per operation is 2.002, comfortably under the proved 3. And the 2,000 resets cost 18,000 units in total — nine each, because 500 increments only ever set bits up to index 8 — against 64,000 for the blind version.

14.8 Let the aggregate bound be $T(n)$, giving every operation the amortized cost $\hat{c} = T(n)/n$. Define

$$\Phi(D_i) \;=\; \sum_{j=1}^{i} (\hat{c} - c_j),$$

with $\Phi(D_0) = 0$ (the empty sum).

Then for each i,

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}) \;=\; c_i + (\hat{c} - c_i) \;=\; \hat{c},$$

because the two sums differ by exactly the single term $\hat{c} - c_i$. So every operation gets amortized cost $\hat{c} = T(n)/n$, which is what the aggregate method gives.

The extra condition. Φ must satisfy $\Phi(D_i) \geq \Phi(D_0) = 0$, that is

$$\sum_{j=1}^{i} c_j \;\le\; i \cdot \hat{c} \qquad \text{for every } i.$$

In words: **the aggregate bound must hold for every prefix of the sequence, not merely for the whole of it.** This is exactly the prefix requirement flagged in Section 14.2.3. For the stack and the counter it is automatic, because the counting arguments there work verbatim on any prefix. A contrived structure whose costs are all crammed into the first few operations would satisfy the aggregate bound at $i = n$ and violate it earlier; for such a structure the aggregate "bound" is not a valid amortized bound at all.

14.9 Take

$$\Phi(T) \;=\; \frac{3}{2}\, num[T] \;-\; \frac{1}{2}\, size[T],$$

that is, $a = 3/2$ and $b = 1/2$.

Sanity checks. Immediately after an expansion by a factor of 3, the table holds $size/3$ items, so $\Phi = 3/2 \cdot \frac{size}{3} - 1/2\, size = \frac{size}{2} - \frac{size}{2} = 0$. When the table is full, $num = size$ and $\Phi = 3/2\, size - 1/2\, size = size$ — exactly the number of items the next expansion will copy. And $\Phi \ge 0$ whenever $num \ge size/3$, which holds because the table is at least one-third full from the moment it is created by an expansion. An empty table has $\Phi = 0$.

No expansion. $c_i = 1$, $size_i = size_{i-1}$, $num_i = num_{i-1} + 1$:

$$\begin{aligned}\hat{c}_i \;&=\; 1 + 3/2\,(num_i - num_{i-1}) - 1/2\,(size_i - size_{i-1}) \;=\; 1 + 3/2 \\ &=\; 5/2.\end{aligned}$$

Expansion. Here $num_{i-1} = size_{i-1}$, $size_i = 3\, size_{i-1}$, $num_i = num_{i-1} + 1$, and $c_i = num_i$:

$$\begin{aligned}\hat{c}_i &= num_i + \left(\tfrac{3}{2} num_i - \tfrac{1}{2} size_i\right) - \left(\tfrac{3}{2} num_{i-1} - \tfrac{1}{2} size_{i-1}\right) \\ &= num_i + \tfrac{3}{2} num_i - \tfrac{3}{2} size_{i-1} - \tfrac{3}{2}(num_i - 1) + \tfrac{1}{2} size_{i-1} \\ &= num_i + \tfrac{3}{2} - size_{i-1} \\ &= (size_{i-1} + 1) + \tfrac{3}{2} - size_{i-1} \\ &= \tfrac{5}{2}.\end{aligned}$$

Both cases give exactly 5/2, so `TABLE-INSERT` is $O(1)$ amortized and n insertions cost at most $2.5n$ — cheaper than doubling's $3n$, at the price of wasting more memory.

General factor $\gamma > 1$. Take

$$\Phi(T) \;=\; \frac{\gamma}{\gamma - 1}\, num[T] \;-\; \frac{1}{\gamma - 1}\, size[T],$$

and the same two computations give an amortized cost of

$$1 + \frac{\gamma}{\gamma - 1}$$

in both cases: 3 for $\gamma = 2$, 2.5 for $\gamma = 3$, and 4 for $\gamma = 1.5$. The bound stays constant for every fixed $\gamma > 1$ and blows up as $\gamma \to 1$ — which is exactly Exercise 14.10 in the limit.

Listing 14.13 measures the $\gamma = 2$ and $\gamma = 3$ policies and the fixed-increment policy side by side.

Listing 14.13 Three growth policies compared: doubling, tripling, and adding a fixed 100 slots. Compile with `g++ -std=c++17 -O2 -o growth growth.cpp`.

```
// Solution to Exercise 14.9 and 14.10.  Three growth policies for a
// dynamic table are run on the same n insertions and the work is counted:
//     double the size,  triple the size,  add a fixed 100 slots.
// The first two are O(1) amortized; the third is Theta(n).
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

struct Growing {
    long long num = 0, size = 0, work = 0, expansions = 0;
    int factor;              // multiply by `factor`, or 0 to add `step`
    long long step;

    Growing(int f, long long s) : factor(f), step(s) {}

    void insert() {
        if (num == size) {
            work += num;                                // copy everything across
            size = (factor > 0) ? (size == 0 ? 1 : size * factor) : size + step;
            ++expansions;
        }
        ++num; work += 1;
    }
};

void run(const string& name, int factor, long long step, long long n) {
    Growing T(factor, step);
    for (long long i = 0; i < n; ++i) T.insert();
    cout << left << setw(22) << name << right
         << " n=" << setw(9) << n
         << " work=" << setw(13) << T.work
         << " work/insert=" << setw(11) << fixed << setprecision(3)
         << (double)T.work / (double)n
         << "  expansions=" << T.expansions
         << "  final size=" << T.size << "\n";
}
```

Listing 14.13 (Continued) *Three growth policies compared: doubling, tripling, and adding a fixed 100 slots. Compile with g++ -std=c++17 -O2 -o growth growth.cpp.*

```
int main() {
    cout << "Each row inserts n items into an initially empty table.\n\n";
    for (long long n : {1000LL, 10000LL, 100000LL, 1000000LL}) {
        run("double the size",  2, 0,   n);
        run("triple the size",  3, 0,   n);
        run("add 100 slots",    0, 100, n);
        cout << "\n";
    }
    cout << "Doubling stays below its bound of 3 units per insert and tripling\n"
            "below its bound of 2.5: both constant.  Adding a fixed 100 slots grows\n"
            "without bound -- about n/200 per insert, which is Theta(n).\n";
    return 0;
}
```

```
Each row inserts n items into an initially empty table.

double the size        n=     1000 work=         2023 work/insert=      2.023
expansions=11  final size=1024
triple the size        n=     1000 work=         2093 work/insert=      2.093  expansions=8
final size=2187
add 100 slots          n=     1000 work=         5500 work/insert=      5.500
expansions=10  final size=1000

double the size        n=    10000 work=        26383 work/insert=      2.638
expansions=15  final size=16384
triple the size        n=    10000 work=        19841 work/insert=      1.984
expansions=10  final size=19683
add 100 slots          n=    10000 work=       505000 work/insert=     50.500
expansions=100  final size=10000

double the size        n=   100000 work=       231071 work/insert=      2.311
expansions=18  final size=131072
triple the size        n=   100000 work=       188573 work/insert=      1.886
expansions=12  final size=177147
add 100 slots          n=   100000 work=     50050000 work/insert=    500.500
expansions=1000  final size=100000

double the size        n=  1000000 work=      2048575 work/insert=      2.049
expansions=21  final size=1048576
triple the size        n=  1000000 work=      1797161 work/insert=      1.797
expansions=14  final size=1594323
add 100 slots          n=  1000000 work=   5000500000 work/insert=   5000.500
expansions=10000  final size=1000000

Doubling stays below its bound of 3 units per insert and tripling
below its bound of 2.5: both constant.  Adding a fixed 100 slots grows
without bound -- about n/200 per insert, which is Theta(n).
```

The doubling and tripling rows hold steady between 1.8 and 2.7 as n grows by three orders of magnitude, always below their proved ceilings of 3 and 2.5. The fixed-increment row multiplies by 10 every time n multiplies by 10: 5.5, 50.5, 500.5, 5000.5. That is the signature of $\Theta(n)$.

14.10 With a fixed increment of $s = 100$ slots, the table starts with no slots and grows to $s, 2s, 3s, \ldots$ Let us count step by step.

When do expansions happen? One happens whenever an insertion finds the table full, that is, when $num = size$. That is the case before insertion 1 (with $num = size = 0$), before insertion $s + 1$ (with $num = size = s$), before insertion $2s + 1$, and so on. So during n insertions there are $m = \lceil n/s \rceil$ expansions, the j-th of them happening when the table holds $(j - 1)s$ items.

What does each cost? The j-th expansion copies the $(j - 1)s$ items already present. The first copies nothing; the second copies s; the last copies $(m - 1)s$.

Add them up. Using $1 + 2 + \cdots + (m - 1) = \frac{m(m-1)}{2}$,

$$\sum_{j=1}^{m} (j - 1)\, s \;=\; s \cdot \frac{m(m-1)}{2}, \qquad m = \left\lceil \frac{n}{s} \right\rceil.$$

Adding the n writes of the items themselves, the **exact** total is

$$T(n) \;=\; n + s \cdot \frac{m(m-1)}{2} \;\approx\; n + \frac{n^2}{2s} \;=\; n + \frac{n^2}{200} \;=\; \Theta(n^2), \tag{14.18}$$

so the amortized cost per insertion is about $n/(2s)$, which is $\Theta(n)$ — **linear, not constant.**

Check Equation (14.18) against the measurement in **Listing 14.13**. At $n = 10^6$, $m = 10^4$ and the formula gives $10^6 + 100 \cdot \frac{10^4 \cdot 9999}{2} = 10^6 + 4{,}999{,}500{,}000 = 5{,}000{,}500{,}000$; the program reports 5,000,500,000. At $n = 10^5$, $m = 1000$ and the formula gives $10^5 + 100 \cdot \frac{1000 \cdot 999}{2} = 50{,}050{,}000$; the program reports 50,050,000. At $n = 1000$ it gives $1000 + 100 \cdot \frac{10 \cdot 9}{2} = 5500$, again the program's figure. Exact.

The reason, once more: the expansion costs form an **arithmetic** series $s, 2s, 3s, \ldots$, whose sum is quadratic, rather than a **geometric** series, whose sum is linear in its largest term.

14.11 Yes, it survives. Any contraction threshold τ with $0 < \tau < 1/2$ works, and $\tau = 1/3$ is such a threshold.

Why there is enough slack. After an expansion the load factor is $1/2$. After a contraction by halving at threshold $1/3$, the table holds just under $size/3$ items in $size/2$ slots, so the new load factor is just under $2/3$ — in particular, at least $1/3$ and at most $2/3$. From either resting point, reaching the next trigger requires a constant fraction of $size$ operations: to reach load factor 1 from $1/2$ takes $size/2$ insertions; to reach load factor $1/3$ from $1/2$ takes $size/6$ deletions. So each $\Theta(size)$ event is preceded by $\Theta(size)$ cheap operations, which is what an amortized bound needs.

The potential function.

$$\Phi(T) \;=\; \begin{cases} 2\,num[T] - size[T] & \text{if } \alpha(T) \geq \frac{1}{2}, \\ size[T] - 2\,num[T] & \text{if } \alpha(T) < \frac{1}{2}. \end{cases}$$

Equivalently, $\Phi(T) = |\,2\,num[T] - size[T]\,|$.

Why that is the right function. It is 0 exactly when the table is half full — the resting state after an expansion — and non-negative everywhere, being an absolute value. It rises to $size$ as the table fills, matching the cost of the expansion. And it rises as the table empties: at the contraction trigger, $num \approx size/3$, so $\Phi \approx size - 2\,size/3 = size/3$, which is exactly the number of items the contraction must copy. In both directions the potential has saved up the cost of the event it can see coming, which is the only thing a potential function has to do.

The general threshold. For a contraction threshold $\tau < 1/2$, take the lower branch to be $\frac{\tau}{\frac{1}{2}-\tau}(1/2\,size - num)$; the coefficient is chosen so that Φ equals the contraction cost $\tau \cdot size$ at the moment the trigger fires. With $\tau = 1/4$ the coefficient is 1, recovering Section 14.7.7; with $\tau = 1/3$ it is 2, recovering the formula above.

14.12 Write α_i, num_i, $size_i$ for the load factor, item count and size after operation i, and recall potential Equation (14.15):

$$\Phi = \begin{cases} 2\,num - size & \text{if } \alpha \geq \frac{1}{2}, \\ \frac{1}{2}size - num & \text{if } \alpha < \frac{1}{2}. \end{cases}$$

`TABLE-INSERT`**, case A:** $\alpha_{i-1} \geq 1/2$**.** Then $\alpha_i \geq \alpha_{i-1} \geq 1/2$ as well, so both potentials use the upper branch and the computation is word for word that of Section 14.7.5: $\hat{c}_i = 3$, whether or not an expansion occurred.

`TABLE-INSERT`**, case B:** $\alpha_{i-1} < 1/2$**.** The table is less than half full, so it is certainly not full, so no expansion occurs and $c_i = 1$, $size_i = size_{i-1} = s$.

Sub-case B1: $\alpha_i < 1/2$ *too.* Both potentials use the lower branch:

$$\hat{c}_i = 1 + \left(\frac{s}{2} - num_i\right) - \left(\frac{s}{2} - num_{i-1}\right) = 1 + (num_{i-1} - num_i) = 1 - 1 = 0.$$

Sub-case B2: $\alpha_{i-1} < 1/2 \leq \alpha_i$. The potential switches branch:

$$\begin{aligned} \hat{c}_i &= 1 + (2\,num_i - s) - \left(\frac{s}{2} - num_{i-1}\right) \\ &= 1 + 2(num_{i-1} + 1) - s - \frac{s}{2} + num_{i-1} \\ &= 3\,num_{i-1} + 3 - \frac{3}{2}s. \end{aligned}$$

Since $\alpha_{i-1} < 1/2$ gives $num_{i-1} < s/2$,

$$\hat{c}_i < 3 \cdot \frac{s}{2} + 3 - 3/2\,s = 3.$$

So `TABLE-INSERT` has $\hat{c}_i \leq 3$ in every case.

`TABLE-DELETE`**, case C:** $\alpha_{i-1} \geq 1/2$**.** The table is at least half full, so a single deletion cannot take it below a quarter (provided $size \geq 4$); no contraction, $c_i = 1$, $size_i = size_{i-1} = s$, $num_i = num_{i-1} - 1$.

Sub-case C1: $\alpha_i \geq 1/2$. Both branches upper:

$$\hat{c}_i = 1 + (2\,num_i - s) - (2\,num_{i-1} - s) = 1 + 2(num_i - num_{i-1}) = 1 - 2 = -1.$$

Sub-case C2: $\alpha_i < 1/2 \leq \alpha_{i-1}$. Branch switch the other way:

$$\begin{aligned} \hat{c}_i &= 1 + \left(\frac{s}{2} - num_i\right) - (2\,num_{i-1} - s) \\ &= 1 + \frac{3}{2}s - num_i - 2(num_i + 1) \\ &= \frac{3}{2}s - 3\,num_i - 1. \end{aligned}$$

From $\alpha_{i-1} \geq 1/2$ we get $s \leq 2\,num_{i-1} = 2\,num_i + 2$, so

$$\hat{c}_i \;\le\; \frac{3}{2}(2\,num_i + 2) - 3\,num_i - 1 \;=\; 3\,num_i + 3 - 3\,num_i - 1 \;=\; 2.$$

TABLE-DELETE, case D: $\alpha_{i-1} < 1/2$. Both potentials use the lower branch.

Sub-case D1: no contraction. $c_i = 1$, $size_i = size_{i-1} = s$:

$$\hat{c}_i \;=\; 1 + \left(\frac{s}{2} - num_i\right) - \left(\frac{s}{2} - num_{i-1}\right) \;=\; 1 + (num_{i-1} - num_i)$$
$$= \; 1 + 1 \;=\; 2.$$

Sub-case D2: a contraction fires. It fires when the deletion leaves $num_i < s/4$; with s a power of two this means $num_i = \frac{s}{4} - 1$, and then $size_i = \frac{s}{2}$ and $c_i = num_i + 1$ (one deletion plus num_i copies).

$$\begin{aligned}
\hat{c}_i &= (num_i + 1) + \left(\frac{s}{4} - num_i\right) - \left(\frac{s}{2} - num_{i-1}\right) \\
&= (num_i + 1) + \frac{s}{4} - num_i - \frac{s}{2} + (num_i + 1) \\
&= num_i + 2 - \frac{s}{4} \\
&= \left(\frac{s}{4} - 1\right) + 2 - \frac{s}{4} \\
&= 1.
\end{aligned}$$

So TABLE-DELETE has $\hat{c}_i \le 2$ in every case. The maximum amortized cost over all eight cases is 3, so any sequence of n insertions and deletions costs at most $3n + \Phi(D_0) = 3n$ starting from an empty table. ■

14.13 The correctness — and that alone is fatal. The cost bound, surprisingly, survives.

Correctness. The items in S_{IN} sit oldest-at-the-bottom. Popping half of them moves the **newest** half onto S_{OUT}, where they land on top and are served first. So the very next Dequeue returns a recent arrival while older items are still buried in S_{IN}. The structure is no longer a queue.

Concretely: `Enqueue 1, 2, 3, 4`, then Dequeue. S_{IN} is [1,2,3,4] bottom to top. Moving half means popping 4 and 3 and pushing them onto S_{OUT}, which becomes [4,3] bottom to top, so the top is 3. The Dequeue returns **3**, when it should return 1.

Cost. Keep the potential $\Phi = 2 \times |S_{IN}|$ and work through a half transfer step by step. Suppose S_{OUT} is empty and S_{IN} holds e items.

1. The transfer moves $e/2$ items, at 2 units each (one pop, one push): e units. The final pop adds 1. So the actual cost is $c_i = e + 1$.
2. Afterward S_{IN} still holds $e/2$ items, so the potential falls from $2e$ to $2 \cdot \frac{e}{2} = e$: a drop of e.
3. The amortized cost is therefore $\hat{c}_i = (e + 1) - e = 1$, exactly as for a full transfer.

Enqueue is still charged 3 and an ordinary Dequeue still 1, so the amortized bound of $O(1)$ per operation survives. The reason is that each item still crosses from S_{IN} to S_{OUT} **at most once** in its lifetime: an item left behind in S_{IN} is not moved twice, it is simply moved later. So the modification is a pure correctness bug — which is worse, not better, because a program that is

fast and wrong gives no warning. **The full transfer is not needed for speed; it is needed for the reversal that makes the order first-in-first-out.**

14.14 Claim. In a disjoint-set forest maintained with union by rank, any tree whose root has rank r contains at least 2^r nodes.

Proof by induction on the number of `Union` operations performed.

Base. Before any union, every tree is a single node with rank 0, and $2^0 = 1$ node. The claim holds.

Inductive step. Assume the claim holds for every tree in the current forest. Consider the next `Union`, uniting trees with roots x and y, of ranks r_x and r_y and sizes $n_x \geq 2^{r_x}$, $n_y \geq 2^{r_y}$ by hypothesis. Union by rank does one of three things.

- If $r_x < r_y$: x is hung under y. The new root is y, its rank is unchanged at r_y, and the new tree has $n_x + n_y > n_y \geq 2^{r_y}$ nodes. Claim holds.
- If $r_x > r_y$: symmetric.
- If $r_x = r_y = r$: one root is hung under the other and the survivor's rank becomes $r + 1$. The new tree has $n_x + n_y \geq 2^r + 2^r = 2^{r+1}$ nodes. Claim holds.

No other operation changes a rank. (Path compression moves nodes but never changes a rank and never removes a node from a tree, so it cannot violate the claim.) ■

Deduction. Let r be the rank of any root. Its tree has at least 2^r nodes, and no tree can have more than n nodes, so $2^r \leq n$ and hence $r \leq \lg n$; since r is an integer, $r \leq \lfloor \lg n \rfloor$. Because rank is an upper bound on height (it is defined to be, and the three cases above preserve that), **no tree has height greater than** $\lfloor \lg n \rfloor$, so `FindSet` follows at most $\lfloor \lg n \rfloor$ pointers. ■

This is confirmed by measurement: in **Listing 14.10**, workload 2 with union by rank only reports a tallest tree of height exactly 15 on $n = 32768 = 2^{15}$ elements.

14.15 Recall $A_0(j) = j + 1$ and $A_k(j) = A_{k-1}^{(j+1)}(j)$ for $k \geq 1$.

Closed forms for the first three levels.

$A_0(j) = j + 1$.

$A_1(j) = A_0^{(j+1)}(j)$ — add 1 to j, a total of $j + 1$ times — $= j + (j + 1) = 2j + 1$.

$A_2(j) = A_1^{(j+1)}(j)$. Applying A_1 once sends $j \mapsto 2j + 1$, which is $j + 1 \mapsto 2(j + 1)$ in "plus one" coordinates; so applying it i times sends $j \mapsto 2^i(j + 1) - 1$. With $i = j + 1$:

$$A_2(j) \;=\; 2^{j+1}(j + 1) - 1.$$

Check: $A_2(1) = 2^2 \cdot 2 - 1 = 7$. ✓

$A_3(1)$**, exactly.**

$$A_3(1) \;=\; A_2^{(2)}(1) \;=\; A_2(A_2(1)) \;=\; A_2(7).$$

And $A_2(7) = 2^8 \cdot 8 - 1 = 256 \cdot 8 - 1 = 2048 - 1 = \boxed{2047}$.

$A_4(1) > 2^{2048}$**.**

$$A_4(1) \;=\; A_3^{(2)}(1) \;=\; A_3(A_3(1)) \;=\; A_3(2047) \;=\; A_2^{(2048)}(2047).$$

The very first of those 2048 applications gives

$$A_2(2047) \;=\; 2^{2048} \cdot 2048 - 1 \;>\; 2^{2048},$$

and every subsequent application only increases the value, since $A_2(j) > j$ for all $j \geq 1$. So $A_4(1) > 2^{2048}$ — in fact vastly larger, since the remaining 2047 applications each exponentiate again.

The conclusion. By definition $\alpha(n) = \min\{k: A_k(1) \geq n\}$. For any $n \leq 2^{2048}$ we have $A_4(1) > 2^{2048} \geq n$, so $k = 4$ already works, and therefore $\alpha(n) \leq 4$.

The observable universe contains roughly 2^{266} atoms. Since $2^{266} \ll 2^{2048}$, **every disjoint-set problem that could ever be represented physically has $\alpha(n) \leq 4$.** For α to reach 5 you would need more than $A_4(1)$ elements, which is not a shortage of engineering but a shortage of universe.

14.16 Listing 14.14 records the capacity after every `push_back`, prints each reallocation with its growth ratio, and totals the copies.

Listing 14.14 Measuring what `std::vector<int>::push_back` really does. Compile with `g++ -std=c++17 -O2 -o vectorgrow vectorgrow.cpp`.

```
// Solution to Exercise 14.16.  What does std::vector actually do when it
// grows?  The program watches capacity() across a million push_back
// calls, records every reallocation, and counts the total number of
// elements copied, comparing it with the 3n bound of Section 14.7.
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

int main() {
    vector<int> v;
    size_t cap = v.capacity();
    long long copied = 0, reallocations = 0;

    cout << "first twelve reallocations:\n";
    cout << "  at push #   old capacity   new capacity   growth ratio\n";
    for (long long i = 1; i <= 1000000; ++i) {
        v.push_back((int)i);
        if (v.capacity() != cap) {
            copied += i - 1;                 // the elements already present
            ++reallocations;
            if (reallocations <= 12)
                cout << setw(11) << i << setw(15) << cap
                     << setw(15) << v.capacity() << setw(15)
                     << fixed << setprecision(3)
                     << (cap == 0 ? 0.0 : (double)v.capacity() / (double)cap) << "\n";
            cap = v.capacity();
        }
    }

    cout << "\nover 1000000 push_back calls:\n";
    cout << "  reallocations              = " << reallocations << "\n";
    cout << "  elements copied in total   = " << copied << "\n";
    cout << "  total element writes       = " << copied + 1000000
         << "  (copies plus the pushes themselves)\n";
    cout << "  writes per push_back       = " << fixed << setprecision(6)
         << (double)(copied + 1000000) / 1000000.0 << "\n";
    cout << "  the 3n bound of Section 14.7 = " << 3 * 1000000 << "\n";
    cout << "  final size = " << v.size() << ", final capacity = " << v.capacity() << "\n";
    return 0;
}
```

```
first twelve reallocations:
  at push #   old capacity   new capacity   growth ratio
          1              0              1          0.000
          2              1              2          2.000
          3              2              4          2.000
          5              4              8          2.000
          9              8             16          2.000
         17             16             32          2.000
         33             32             64          2.000
         65             64            128          2.000
        129            128            256          2.000
        257            256            512          2.000
        513            512           1024          2.000
       1025           1024           2048          2.000

over 1000000 push_back calls:
  reallocations              = 21
  elements copied in total   = 1048575
  total element writes       = 2048575  (copies plus the pushes themselves)
  writes per push_back       = 2.048575
  the 3n bound of Section 14.7 = 3000000
  final size = 1000000, final capacity = 1048576
```

Everything the chapter predicted is visible.

The reallocations happen at pushes 1,2,3,5,9,17,33,65, … — that is, at $2^j + 1$ — which is precisely the expansion condition of Section 14.7.2: **insertion i triggers an expansion exactly when $i - 1$ is a power of 2.**

The growth ratio is exactly 2.000 every time: this implementation doubles, as `TABLE-INSERT` of **Figure 14.16** does. The C++ standard [12] requires only that `push_back` take amortized constant time; the factor 2 is this library's choice, and other libraries use 1.5, which Exercise 14.9 shows is equally valid.

A million pushes cause only **21** reallocations, and copy 1,048,575 elements in total, which is $2^{20} - 1$ — one less than the capacity at the end, exactly as the geometric sum $1 + 2 + 4 + \cdots + 2^{19} = 2^{20} - 1$ requires.

Total element writes: 2,048,575, or 2.049 per `push_back`, against the proved ceiling of 3. Compare with **Listing 14.8**, which measured 2,048,575 for the same n from an independent implementation. The standard library and the textbook agree to the unit.

14.17 The sequence. Let k be the counter's width and let n be even. Set the counter to $2^{k-1} - 1$, whose binary representation is $k - 1$ ones. Now alternate:

$$\text{Increment, Decrement, Increment, Decrement, } \ldots$$

- The `Increment` takes $0\underbrace{11\cdots1}_{k-1}$ to $1\underbrace{00\cdots0}_{k-1}$: it resets $k - 1$ ones and sets one bit, costing k.
- The `Decrement` takes $1\underbrace{00\cdots0}_{k-1}$ back to $0\underbrace{11\cdots1}_{k-1}$: it must borrow through $k - 1$ zeros, setting each to 1, and clear the top bit, costing k.

Every operation costs k, so n operations cost nk. With $k = \lg n$ that is $\Theta(n\lg n)$, so the amortized cost per operation is $\Theta(k)$, **not** $O(1)$. (If the counter must start at zero, the same thing happens even more simply: one `Decrement` from zero wraps round to all ones at cost k, the next `Increment` wraps back to zero at cost k, and alternating the two flips all k bits every time.)

Why Φ = number of 1 bits fails. Compute the amortized cost of the `Decrement` above. Its actual cost is $c_i = k$. Before it, the counter has 1 one bit; after it, $k - 1$ one bits. So

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}) \;=\; k + (k-1) - 1 \;=\; 2k - 2,$$

which grows with k rather than being bounded. The potential *rises* by almost k at the same moment the operation does k units of work, so instead of canceling, the two terms add.

The deeper reason. The potential $= $ (number of 1 bits) works for `Increment` alone because `Increment` is the only operation, and the only way to accumulate 1 bits is to pay 2 for each. Adding `Decrement` gives the adversary a way to *create* 1 bits cheaply relative to the stored potential — and then destroy them again — so the structure can be driven back and forth across an expensive boundary indefinitely. It is the counter's version of the table thrashing in Section 14.7.6, and the diagnosis is the same: **there is no slack between the expensive state and its neighbor.**

(For the record, a counter *can* support both operations in $O(1)$ amortized time, but not in this representation. Redundant binary representations, which allow the digit 2 as well as 0 and 1, do support increment and decrement in $O(1)$ *worst-case* time — which is a good illustration of the

point in Section 14.11 that when an amortized bound is not enough, the answer is usually a different representation rather than a cleverer analysis.)

References

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

[2] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[3] R. E. Tarjan, "Amortized computational complexity," *SIAM Journal on Algebraic and Discrete Methods*, vol. 6, no. 2, pp. 306–318, Apr. 1985.

[4] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM*, vol. 32, no. 3, pp. 652–686, Jul. 1985.

[5] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[6] B. A. Galler and M. J. Fisher, "An improved equivalence algorithm," *Communications of the ACM*, vol. 7, no. 5, pp. 301–303, May 1964.

[7] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[8] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, Apr. 1975.

[9] W. Ackermann, "Zum Hilbertschen Aufbau der reellen Zahlen" ["On Hilbert's construction of the real numbers"], *Mathematische Annalen*, vol. 99, no. 1, pp. 118–133, Dec. 1928.

[10] R. E. Tarjan, *Data Structures and Network Algorithms*. Philadelphia, PA, USA: SIAM, 1983.

[11] M. L. Fredman and M. E. Saks, "The cell probe complexity of dynamic data structures," in *Proc. 21st Annual ACM Symposium on Theory of Computing (STOC)*, Seattle, WA, USA, 1989, pp. 345–354.

[12] ISO/IEC 14882:2017, *Programming Languages — C++*, International Organization for Standardization, Geneva, Switzerland, 2017.

Chapter 15
All-Pairs Shortest Paths

By the end of this chapter you will be able to take a weighted directed graph and produce, in one table, the cheapest route between *every* ordered pair of its vertices — not one journey, but all of them at once. You will be able to say exactly what the naive approach costs, and why it is sometimes still the right answer. You will be able to derive, from nothing but the definition of a shortest path, the recurrence $l_{ij}^{(m)} = \min(l_{ij}^{(m-1)}, \min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\})$, recognize in it the shape of ordinary matrix multiplication, and exploit that resemblance twice: once to get an algorithm, and once more — through repeated squaring — to make it faster. You will be able to state, derive, implement, trace and prove correct the Floyd–Warshall algorithm, and say precisely why the loop over k must be the outermost of the three. You will be able to reconstruct an actual route, not merely its cost, from a predecessor matrix Π. You will be able to compute the transitive closure of a directed graph by changing two operators and nothing else. You will be able to reweight a graph with negative edges so that Dijkstra becomes legal again, prove that the reweighting changes no shortest path, and thereby implement Johnson's algorithm for sparse graphs. And you will be able to answer, when someone hands you a graph and asks which of these four algorithms to run, with a number rather than an opinion.

15.1 Where We Left Off

The standard reference treatment is Chapter 23 of Cormen, Leiserson, Rivest and Stein [1] (Chapter 25 in the third edition, whose figure numbers several captions below mention); the original papers are Floyd's half-page note of 1962 [2], Warshall's boolean-matrix paper of the same year [3], and Johnson's 1977 reweighting paper [4]. Kleinberg and Tardos [5] give a particularly readable account of the Bellman–Ford side of the story, and Sedgewick and Wayne [6] of the implementation side.

15.1.1 What Chapter 13 gave us

Chapter 13 solved the **single-source shortest-paths problem**. The setting was a weighted directed graph $G = (V, E)$ together with a **weight function** $w: E \to \mathbb{R}$ that attaches a real number to every edge. The **weight of a path** $p = \langle v_0, v_1, \dots, v_k\rangle$ is the sum of the weights of the edges along it,

$$w(p) = \sum_{i=1}^{k} w\,(v_{i-1}, v_i),$$

and the **shortest-path weight** from u to v is

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \rightsquigarrow v \text{ along } p\} & \text{if such a path exists,} \\ \infty & \text{otherwise.} \end{cases}$$

The symbol $\rightsquigarrow$ means "there is a path from … to …", and δ (the Greek letter *delta*) is reserved throughout this book for the *true* shortest-path weight — the answer, the thing we are trying to compute. Every algorithm in Chapter 13 worked by keeping an **estimate** $d[v]$, a number that starts too large and is repeatedly pushed down, together with a **predecessor** $\pi[v]$, the vertex from which v is currently best reached. The single operation that pushed the estimate down was **relaxation**:

> **DEFINITION (recalled from Chapter 13).** To **relax** an edge (u, v) is to ask whether going to v via u is better than the best route to v found so far, and to record the improvement if it is:

$$\text{if } d[u] + w(u, v) < d[v] \text{ then } d[v] \leftarrow d[u] + w(u, v) \text{ and } \pi[v] \leftarrow u.$$

> Every correct shortest-path algorithm in this book is a strategy for deciding *which* edge to relax next, and *how many times*.

Chapter 13 gave three such strategies:

- **Dijkstra's algorithm**, which requires every weight to be non-negative, extracts vertices from a min-priority queue in increasing order of d, and relaxes each edge exactly once. With a linear array as the queue it costs $O(V^2 + E)$; with a binary min-heap, $O(E\lg V)$; with a Fibonacci heap, $O(V\lg V + E)$.
- **Bellman–Ford**, which tolerates negative weights, relaxes *every* edge $|V| - 1$ times, at a cost of $O(VE)$, and detects a negative-weight cycle by testing whether a further relaxation is still possible.
- The **DAG algorithm**, which topologically sorts an acyclic graph and relaxes in that order, at a cost of $\Theta(V + E)$.

Chapter 13 also proved two facts we shall lean on constantly. The first is the **optimal-substructure theorem**, proved there as Theorem 13.1. In the third edition of Cormen et al. it is Lemma 24.1; in the fourth edition [1] it is Lemma 22.1. Whichever label you meet it under, it says:

> **THEOREM 13.1, RECALLED (subpaths of shortest paths are shortest paths; Lemma 22.1 of [1]).** Let $p = \langle v_0, v_1, \dots, v_k \rangle$ be a shortest path from v_0 to v_k, and for any $0 \le a \le b \le k$ let $p_{ab} = \langle v_a, v_{a+1}, \dots, v_b \rangle$ be the piece of p between v_a and v_b. Then p_{ab} is a shortest path from v_a to v_b.

The proof is one line of arithmetic and a contradiction: decompose $w(p) = w(p_{0a}) + w(p_{ab}) + w(p_{bk})$. If some path p'_{ab} from v_a to v_b were lighter than p_{ab}, then splicing it in would give a path from v_0 to v_k of weight $w(p_{0a}) + w(p'_{ab}) + w(p_{bk}) < w(p)$, contradicting the assumption that p was shortest. This lemma is the entire reason dynamic programming applies to shortest paths at all, and it will be used, in two different disguises, in §15.4 and §15.6.

The second fact concerns **negative-weight cycles**. If a cycle of negative total weight lies on a route from u to v, then going round it once more lowers the cost, and going round it a thousand times lowers it further; there is no minimum, and $\delta(u, v)$ is $-\infty$. Everything in this chapter assumes the input has **no negative-weight cycle**. Negative *edges* are welcome — this chapter's running example has two — but negative *cycles* are not. §15.6.14 shows how to detect them for free.

15.1.2 The new question

Chapter 13 answered "how do I get from here to everywhere?" This chapter answers "how does everyone get to everyone?" Stated precisely:

DEFINITION (the all-pairs shortest-paths problem). Given a weighted directed graph $G = (V, E)$ with weight function $w: E \to \mathbb{R}$, find, for **every** ordered pair of vertices $u, v \in V$, a shortest path from u to v and its weight $\delta(u, v)$. The output is naturally **tabular**: the entry in u's row and v's column is the weight of a shortest path from u to v.

Notice the word *ordered*. In a directed graph the run from u to v and the run from v to u are different journeys with, in general, different costs. A one-way street system makes this obvious; so does an airline timetable. So the output table has $|V|^2$ entries, not $\binom{|V|}{2}$, and it is not symmetric.

Why would anyone want all of them? Because a great many practical questions are questions about the whole table rather than about one entry.

- **Where should the depot go?** To answer that, you need the distance from every candidate site to every customer. One source is not enough; you do not know in advance which source you will choose.
- **What is the diameter of this network?** The diameter is the largest of all the $\delta(u, v)$ — the worst journey in the system. You cannot find the largest of a set without computing the set.
- **Which two pages on this site are furthest apart in clicks?** Same question, different network.
- **Is this relation transitive, and if not, what is its transitive closure?** §15.7 shows that this is the all-pairs problem with two operators swapped.
- **Preprocessing.** If a table will be queried millions of times, computing it once and looking entries up in $O(1)$ beats re-running a single-source algorithm per query, provided the graph fits in memory.

From here on we drop the bars inside asymptotic notation, exactly as Chapter 13 did: $O(V^3)$ means $O(|V|^3)$. And because this chapter works with matrices, we give the number of vertices the short name n, so $n = |V|$ throughout.

15.2 The All-Pairs Problem, Precisely

15.2.1 The input: an adjacency matrix

The single-source algorithms of Chapter 13 took the graph as **adjacency lists**, because they walked outward from one vertex and never needed to ask "is there an edge from i to j?" out of the blue. The algorithms of this chapter do ask that question, constantly, and they produce a matrix as output. It is therefore natural to take the input as a matrix too.

Number the vertices $1, 2, \ldots, n$. (Any numbering will do; the algorithms do not care which, although §15.6.6 will show that the *order* matters enormously to *how* Floyd–Warshall works, even though it does not change the answer.) The input is then an $n \times n$ matrix $W = (w_{ij})$, pictured in **Figure 15.1**.

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 definition is

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ \text{the weight of directed edge } (i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases} \tag{15.1}$$

Three lines, and each one earns its place.

Why $w_{ii} = 0$. The cost of staying where you are is nothing. This is not a convention adopted for tidiness; it is used as a *load-bearing fact* twice in this chapter — once in §15.4.5, where $w_{jj} = 0$ lets us collapse a two-part minimum into one, and once in §15.6.14, where a diagonal entry that has become negative is the proof that a negative-weight cycle exists.

Why ∞ for a missing edge. Because $\min(x, \infty) = x$ and $\infty + c = \infty$, the symbol ∞ behaves in exactly the way "you cannot get there that way" ought to behave. It never wins a minimum against a real number, and it poisons any sum it enters. In a program we represent it by a large integer — every listing in this chapter uses `const long long INF = 1000000000LL;` — and we must then be careful never to compute `INF + INF`, which in a 32-bit `int` would overflow to a negative number and be mistaken for a very good route. Every listing guards the addition explicitly.

Why $i \neq j$ in the second line. A self-loop (i, i) of positive weight is useless: it can only make a route longer. A self-loop of negative weight is a negative-weight cycle, which we have excluded. So $w_{ii} = 0$ is both safe and right.

For the whole of this chapter the running example is the five-vertex graph of **Figure 15.2**. Its edges are $1 \to 2$ of weight 3, $1 \to 3$ of weight 8, $1 \to 5$ of weight -4, $2 \to 4$ of weight 1, $2 \to 5$ of weight 7, $3 \to 2$ of weight 4, $4 \to 1$ of weight 2, $4 \to 3$ of weight -5, and $5 \to 4$ of weight 6.

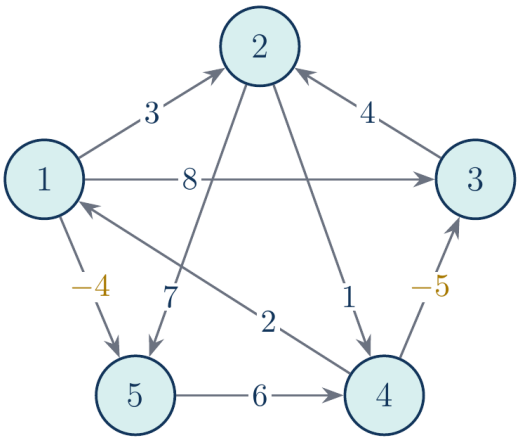


Figure 15.2 The running example: a five-vertex directed graph with nine edges, two of them negative ($1 \to 5$ of weight -4 and $4 \to 3$ of weight -5). It has no negative-weight cycle (§15.6.14 verifies this by machine), so every $\delta(i, j)$ is finite.

Applying Equation (15.1) vertex by vertex gives

$$W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}.$$

Read the first row across: from vertex 1 you can stay put at cost 0; reach 2 directly at cost 3; reach 3 directly at cost 8; there is no edge $1 \to 4$, so ∞; and reach 5 at cost -4. Read the third row: from vertex 3 there is exactly one outgoing edge, to vertex 2, of weight 4; everything else in that row except the diagonal is ∞. Note again that W is **not symmetric**: $w_{15} = -4$ but $w_{51} = \infty$, because the edge $1 \to 5$ exists and $5 \to 1$ does not.

Storing W takes $\Theta(n^2)$ space whether the graph has nine edges or nine million. For the graphs in this chapter that is exactly the right trade, because the *output* is $\Theta(n^2)$ anyway; you cannot report all-pairs distances in less space than it takes to write them down. Johnson's algorithm in §15.8 is the one exception worth knowing: it reads the graph from adjacency lists and only materializes the $n \times n$ answer at the end.

15.2.2 The output: two matrices, not one

We must be precise about what to produce. The first output is the matrix of distances. We call it $D = (d_{ij})$, an $n \times n$ matrix in which d_{ij} holds the weight of a shortest path from vertex i to vertex j, so that at termination

$$d_{ij} = \delta(i, j) \quad \text{for all } i, j \in V. \tag{15.2}$$

(The matrix-multiplication method of §15.4 builds up a sequence of matrices it calls $L^{(m)}$ rather than D, for a reason that will become clear; the last L is the same D.)

The second output is the one beginners most often forget to ask for, and the one that makes the answer *usable*. Knowing that the cheapest run from Keputih to Darmo costs eighteen minutes is of limited help to a driver who wants to know which roads to take. So alongside D we compute the **predecessor matrix**

DEFINITION. The **predecessor matrix** is the $n \times n$ matrix $\Pi = (\pi_{ij})$ in which π_{ij} is NIL if $i = j$ or if there is no path at all from i to j, and otherwise is **the predecessor of j on some shortest path from i**. That is: π_{ij} names the vertex you were standing on immediately before you arrived at j, on one particular cheapest route that began at i.

Read that definition once more, slowly, because the indices are easy to misread. π_{ij} is *not* the second vertex of the route; it is the *second-to-last*. Row i of Π describes the whole tree of cheapest routes out of i, stored by pointing every vertex backward toward i. This is called the **predecessor subgraph of G for i**, defined by

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\},$$

$$E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} - \{i\}\},$$

so that $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$. In words: the vertices of $G_{\pi,i}$ are i itself together with everything reachable from i, and its edges are one per reachable vertex, each pointing from that vertex's recorded predecessor to the vertex itself. If the algorithm is correct then $G_{\pi,i}$ is a **shortest-paths tree rooted at i** — exactly the object Chapter 13 built for a single source, except that we now build n of them simultaneously, one per row of Π.

IN PLAIN LANGUAGE. D tells you *how much*. Π tells you *how*. D is a price list; Π is a pile of n road maps, one per starting point, each folded up into a single row of n numbers. Both are $\Theta(n^2)$ in size, which is why we can afford to keep them both.

15.2.3 Printing a path out of Π

Turning a row of Π back into a route is a three-line recursion, shown in **Figure 15.3**.

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
if i == j
    print i
elseif π_ij == NIL
    print "no path from" i "to" j "exists"
else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
    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.

The trick is that the recursive call comes *before* the print. To print the route from i to j, first print the route from i to j's predecessor, and only then print j. The recursion therefore walks backward from j to i as it descends, and prints forward from i to j as it returns. The three cases are exactly the three cases of the definition of π_{ij}:

1. $i = j$**.** We have arrived back at the start. Print i and stop. This is the base case, and it is the only one that terminates the recursion normally.
2. $\pi_{ij} = \text{NIL}$ **and** $i \neq j$**.** There is no path. Say so and stop.

3. **Otherwise.** Recurse on (Π, i, π_{ij}) — that is, print the route from i up to the vertex just before j — and then print j.

The procedure runs in time linear in the number of vertices on the path it prints, which is at most n, so printing all n^2 paths costs $O(n^3)$: the same as computing them. We shall see it work in §15.6.11, on real numbers produced by a real program.

15.2.4 A first sanity check on sizes

Before we write a single algorithm, it is worth fixing the scale of the problem in mind, because it explains every design decision that follows.

The output has n^2 entries. Even reading it back costs $\Omega(n^2)$. So no all-pairs algorithm can beat $\Omega(n^2)$, and the algorithms of this chapter land at n^3 or a little better — within a factor of n of the absolute floor. For $n = 1000$ that is a billion operations for the answer and a million numbers to store: a second or two, and eight megabytes. For $n = 10{,}000$ it is a trillion operations and eight hundred megabytes: minutes and a large machine. For $n = 100{,}000$ — a modest website — the *output alone* is ten billion numbers, eighty gigabytes, and the all-pairs problem is simply the wrong question. Knowing where that wall stands is part of knowing the material.

15.3 The Obvious Algorithm, and Its Price

Here is the first algorithm anyone thinks of, and it is worth stating first precisely so that we can measure everything else against it:

> **THE NAIVE APPROACH.** To solve all-pairs shortest paths, run a single-source shortest-paths algorithm $|V|$ times, once with each vertex as the source, and stack the $|V|$ answer vectors into a matrix.

This is not a joke and it is not a straw man. It is correct, it is simple, and for some graphs it is the fastest thing known. What it costs depends entirely on which single-source algorithm we are allowed to use, and that depends on the weights.

15.3.1 All weights non-negative: $|V| \times$ Dijkstra

If no edge is negative, Dijkstra applies, and we pay its cost n times over. There are three variants, one per priority-queue implementation, and they are worth separating carefully because the wrong one is routinely quoted.

A linear array as the min-priority queue. Each EXTRACT-MIN scans the array in $O(V)$ time and there are V of them, so extraction costs $O(V^2)$; each DECREASE-KEY is a single assignment, $O(1)$, and there are at most E of them. One run therefore costs $O(V^2 + E)$, and V runs cost

$$O(V \cdot (V^2 + E)) = O(V^3 + VE) = O(V^3),$$

the last step because $E \leq V^2$ always, so $VE \leq V^3$ and the V^3 term dominates. The first variant therefore costs $O(V^3 + VE) = O(V^3)$.

A binary min-heap. Now EXTRACT-MIN costs $O(\lg V)$ and DECREASE-KEY costs $O(\lg V)$, giving one run in $O((V + E)\lg V) = O(E\lg V)$ for a connected graph, and V runs in

$$O(VE\lg V).$$

This second variant comes with a crucial qualifier: it is an **improvement if the graph is sparse**. On a dense graph, where $E = \Theta(V^2)$, this becomes $O(V^3 \lg V)$, which is *worse* than the array version by a factor of $\lg V$. The heap is not universally better; it is better exactly when $E = o(V^2/\lg V)$.

A Fibonacci heap [7]. DECREASE-KEY drops to $O(1)$ amortized while EXTRACT-MIN stays at $O(\lg V)$ amortized, so one run costs $O(V \lg V + E)$ and V runs cost

$$O(V^2 \lg V + VE).$$

This third variant gives the bound Johnson's algorithm inherits in §15.8. It is never worse than either of the other two, though the constant factor hidden in a Fibonacci heap is large enough that in practice a binary heap usually wins below a few hundred thousand vertices.

15.3.2 Negative edges present: $|V| \times$ Bellman–Ford

The moment one edge is negative, Dijkstra becomes illegal: we *cannot* use Dijkstra's algorithm and must run the slower Bellman–Ford algorithm once from each vertex. One run of Bellman–Ford costs $O(VE)$, so V runs cost

$$O(V^2 E),$$

which on a dense graph, where $E = \Theta(V^2)$, is

$$O(V^4).$$

That is the number to remember, and it is genuinely bad. At $n = 500$ it is 6×10^{10} edge relaxations — minutes of computation where Floyd–Warshall would take a fraction of a second. The natural question is: how do we make it better?

15.3.3 Why we cannot simply use Dijkstra anyway

It is tempting to try it anyway. Dijkstra's correctness rests on one assumption: when a vertex u is extracted from the queue with the smallest estimate, that estimate is already final, because any other route to u would have to pass through a vertex still in the queue, whose estimate is at least as large, and then travel further along edges of non-negative weight — so it could only be worse. A negative edge destroys that argument in one stroke: traveling *further* can now make a route *cheaper*.

It is worth seeing the failure rather than merely being told about it, and it is worth seeing something subtler too. **Listing 15.1** runs textbook Dijkstra from every vertex of two graphs — the five-vertex running example, and a four-vertex graph built specifically as a trap — and audits every answer against Floyd–Warshall.

Listing 15.1 Dijkstra audited against Floyd–Warshall on two graphs with negative edges and no negative cycles. Compile with `g++ -std=c++17 -O2 -o dijkstra_fails dijkstra_fails.cpp`.

```
// Chapter 15, Listing: why |V| x Dijkstra is not an option once an edge is negative.
// Dijkstra is run from every vertex and its answers are compared with the ones
// Floyd-Warshall produces. Neither graph has a negative-weight cycle.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

const long long INF = 1000000000LL;
typedef vector<vector<long long>> Matrix;

// Textbook Dijkstra: a vertex is extracted once, declared final, and its outgoing
// edges are relaxed once. That single assumption is what negative edges break.
vector<long long> dijkstra(const Matrix& W, int s) {
    int n = (int)W.size();
    vector<long long> d(n, INF);
    vector<char> done(n, 0);
    d[s] = 0;
    for (int t = 0; t < n; ++t) {
        int u = -1;
        for (int v = 0; v < n; ++v)
            if (!done[v] && d[v] < INF && (u == -1 || d[v] < d[u])) u = v;
        if (u == -1) break;
        done[u] = 1;
        for (int v = 0; v < n; ++v)
            if (!done[v] && W[u][v] < INF && d[u] + W[u][v] < d[v])
                d[v] = d[u] + W[u][v];               // finalized vertices are not
revisited
    }
    return d;
}

Matrix floydWarshall(Matrix D) {
    int n = (int)D.size();
    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][j] = min(D[i][j], D[i][k] + D[k][j]);
    return D;
}
```

Listing 15.1 (Continued) *Dijkstra audited against Floyd–Warshall on two graphs with negative edges and no negative cycles. Compile with g++ -std=c++17 -O2 -o dijkstra_fails dijkstra_fails.cpp.*

```
int audit(const string& title, const Matrix& W) {
    int n = (int)W.size();
    Matrix D = floydWarshall(W);
    cout << title << '\n';
    int wrong = 0;
    for (int s = 0; s < n; ++s) {
        vector<long long> d = dijkstra(W, s);
        for (int v = 0; v < n; ++v)
            if (d[v] != D[s][v]) {
                ++wrong;
                cout << "   delta(" << s + 1 << "," << v + 1 << "): Dijkstra says "
                     << d[v] << ", the truth is " << D[s][v] << "  <-- WRONG\n";
            }
    }
    cout << "   entries Dijkstra got wrong: " << wrong << " out of " << n * n << "\n\n";
    return wrong;
}

int main() {
    Matrix sample = {
        {  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}
    };
    // 1 -> 2 (1), 1 -> 3 (2), 3 -> 2 (-3), 2 -> 4 (1). Vertex 2 is finalized at
    // distance 1 and its edge to 4 relaxed, before vertex 3 lowers 2 to -1.
    Matrix trap = {
        {  0,   1,   2, INF},
        {INF,   0, INF,   1},
        {INF,  -3,   0, INF},
        {INF, INF, INF,   0}
    };
    audit("(a) the five-vertex running example:", sample);
    audit("(b) a four-vertex graph built to expose the flaw:", trap);
    cout << "Moral: on (a) Dijkstra happened to be right, and that is worse than\n"
            "being wrong, because it invites you to trust it on (b).\n";
    return 0;
}
```

Its output:

```
(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).
```

Look carefully at what happened. On the running example, Dijkstra produced all twenty-five correct answers. It was not entitled to; it was lucky. On graph (b) — edges $1 \to 2$ of weight 1, $1 \to 3$ of weight 2, $3 \to 2$ of weight -3, $2 \to 4$ of weight 1 — it fails. Starting at vertex 1 it sets $d[2] = 1$ and $d[3] = 2$, extracts vertex 2 first because $1 < 2$, declares it final, and relaxes $2 \to 4$ to get $d[4] = 2$. Only afterward does it extract vertex 3 and discover that $1 \to 3 \to 2$ costs $2 + (-3) = -1$, which is better than 1. But vertex 2 has already been finalized, its outgoing edge already

relaxed, and the improvement never reaches vertex 4. The true answer is $\delta(1,4) = 1 \to 3 \to 2 \to 4 = 2 - 3 + 1 = 0$; Dijkstra reports 2.

> **THE TRAP.** An algorithm that is wrong *sometimes* is more dangerous than one that is wrong always, because testing will not find it. Do not use Dijkstra on a graph that may contain a negative edge, however well it seems to behave on your sample data. §15.8 shows the legitimate way to get Dijkstra's speed on such a graph: change the weights first, and prove that the change is harmless.

15.3.4 What we are trying to beat

Table 15.1 collects the naive costs. Everything in the rest of this chapter is measured against this table.

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	$O(V^3 + VE)$	$O(V^3)$
all weights ≥ 0, binary heap	Dijkstra	$O(VE\lg V)$	$O(V^3\lg V)$
all weights ≥ 0, Fibonacci heap	Dijkstra	$O(V^2\lg V + VE)$	$O(V^3)$
some weight < 0	Bellman–Ford	$O(V^2E)$	$O(V^4)$

Two observations set up everything that follows. First, on a dense graph with non-negative weights, $V \times$ Dijkstra already achieves $O(V^3)$ — so any new algorithm must either match that and handle negative edges too, or beat it on sparse graphs. Floyd–Warshall does the first; Johnson does the second. Second, the disaster is the last row: $O(V^4)$ for the general case. The matrix-multiplication method of §15.4 will get that down to $\Theta(n^3\lg n)$ and Floyd–Warshall to $\Theta(n^3)$, both of them handling negative edges without complaint.

15.4 Shortest Paths and Matrix Multiplication

15.4.1 The dynamic-programming recipe, recalled

Recall the four-step recipe for a dynamic-programming algorithm that Chapter 7 laid down:

1. **Characterize the structure of an optimal solution.**
2. **Recursively define the value of an optimal solution.**
3. **Compute the value of an optimal solution in a bottom-up fashion.**
4. **Construct an optimal solution from the computed information.**

We shall follow those four steps in order, and then follow them a *second* time in §15.6 with a different characterization in step 1 — which is the whole reason there are two different algorithms in this chapter rather than one. Dynamic programming is not a single algorithm; it is a method, and the method has room for more than one good answer.

15.4.2 Step 1: the structure of a shortest path

The characterization in this section is by **number of edges**. **Figure 15.4** sets it out.

- 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.

Here is the argument in full. Let p be a shortest path from vertex i to vertex j containing **at most m edges**. Since the graph has no negative-weight cycle, m is finite: a shortest path never needs to repeat a vertex, so it has at most $n-1$ edges. Now split into two cases.

Case A: $i = j$. Then the empty path — no edges at all — has weight 0, and since there is no negative-weight cycle, no closed walk from i back to i can do better than 0. So p has weight 0 and no edges.

Case B: $i \neq j$. Then p has at least one edge. Write k for the vertex immediately before j on p, and decompose

$$i \overset{p'}{\rightsquigarrow} k \to j,$$

where p' is everything except the last edge. Because p had at most m edges and we removed one, p' contains **at most $m-1$ edges**. By Theorem 13.1, p' is a shortest path from i to k. And therefore

$$\delta(i,j) = \delta(i,k) + w_{kj}.$$

That last equation is the entire content of step 1. It says: *the cost of the best route from i to j is the cost of the best route to somewhere one step short of j, plus that last step.* We do not know which vertex k is — but there are only n candidates, and we can try them all.

> **IN PLAIN LANGUAGE.** Imagine you are handed the cheapest way to get from Keputih to Darmo, and told it takes at most six road segments. Cover up the last segment with your thumb. Whatever is left must be the cheapest way to get from Keputih to wherever the thumb starts — because if it were not, you could substitute something cheaper and beat a route you were told was already the cheapest. That is Theorem 13.1, and it is the only idea in this section.

15.4.3 Step 2: the recursive definition of $l_{ij}^{(m)}$

Now we name the quantity the recursion will compute; **Figure 15.5** introduces it.

- 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. Let $l_{ij}^{(m)}$ be the **minimum weight of any path from vertex i to vertex j that contains at most m edges**. If no such path exists, $l_{ij}^{(m)} = \infty$.

Every part of that notation matters, and it is easy to get wrong, so let us be exact. The letter l stands for *length* in the sense of weight. The **subscripts** i and j are the endpoints: i first, the source; j second, the destination. The **superscript** (m), written in parentheses, is the *edge budget* — the largest number of edges the path is allowed to use. It is a label, not a power: $l_{ij}^{(2)}$ does not mean l_{ij} squared. Writing it as `l(m)ij` or `lij^m` destroys exactly the distinction that makes the recurrence readable, and the parenthesized superscript is the standard notation in the literature [1].

The base case is $m = 0$. With a budget of zero edges you cannot go anywhere, so the only journeys available are the ones that were already over before they began:

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j. \end{cases} \tag{15.3}$$

In matrix form, $L^{(0)}$ is the matrix with 0 down the diagonal and ∞ everywhere else. Keep that matrix in mind: §15.4.9 will show that it is the *identity element* of the strange multiplication we are about to define, in exactly the way that the ordinary identity matrix — 1 on the diagonal, 0 elsewhere — is the identity for ordinary multiplication.

15.4.4 The recurrence, term by term

For $m \geq 1$ we compute $l_{ij}^{(m)}$ from the values $l_{ij}^{(m-1)}$ already known. **Figure 15.6** gives the recurrence.

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 $m = n-1$.

The recurrence is

$$l_{ij}^{(m)} = \min\Big(l_{ij}^{(m-1)},\ \min_{1\le k\le n}\Big\{l_{ik}^{(m-1)} + w_{kj}\Big\}\Big). \tag{15.4}$$

Take it apart piece by piece, because every symbol is doing a job.

The outer min has two arguments, corresponding to the two things a best-route-with-at-most-m-edges could be.

The first argument, $l_{ij}^{(m-1)}$, is the weight of a shortest path from i to j that uses at most $m-1$ edges. Why is it a candidate? Because "at most m edges" includes "at most $m-1$ edges". Raising the budget can never make things worse, and the best route under the smaller budget is still legal under the larger one. This argument covers the case where the extra edge we were granted turns out to be unnecessary.

The second argument, $\min_{1\le k\le n}\Big\{l_{ik}^{(m-1)} + w_{kj}\Big\}$, covers the case where we do use the extra edge. It is itself a minimum, taken over all n possible choices of the vertex k that we pass through immediately before arriving at j. For each candidate k the cost is $l_{ik}^{(m-1)}$ — the best way to reach k using at most $m-1$ edges — plus w_{kj}, the weight of the single final hop from k to j. If there is no edge $k \to j$ then $w_{kj} = \infty$ and that candidate loses automatically; if k is unreachable from i within $m-1$ edges then $l_{ik}^{(m-1)} = \infty$ and it loses too. The notation $\min_{1\le k\le n}$ means "take the smallest of these n numbers, as k runs from 1 to n"; the subscript under min names the variable being swept and its range, exactly as the subscript under $\sum$ does.

This is precisely the decomposition of §15.4.2, turned into a formula. Step 1 told us that either $i = j$ and the path is empty, or the path ends with some edge $k \to j$ and the part before it is optimal. The recurrence tries the empty extension and all n final edges, and keeps the best.

15.4.5 Why the first argument can be thrown away

Next comes a simplification that looks like a trick and is really an observation about the diagonal of W. Take the inner minimum and consider the particular candidate $k = j$. Its value is

$$l_{ij}^{(m-1)} + w_{jj} = l_{ij}^{(m-1)} + 0 = l_{ij}^{(m-1)},$$

because $w_{jj} = 0$ for all j, by the first line of Equation (15.1). But $l_{ij}^{(m-1)}$ is exactly the first argument of the outer minimum. So the first argument is *already one of the candidates* in the inner minimum, and including it separately changes nothing. The recurrence collapses to

$$l_{ij}^{(m)} = \min_{1 \le k \le n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}, \qquad \text{note that } w_{jj} = 0 \text{ for all } j. \tag{15.5}$$

> **WHY THIS MATTERS.** The simplified form has exactly one minimum over exactly one index. That is the shape of a matrix product, and recognizing it is the point of this whole section. Had we defined w_{jj} to be anything other than 0 — ∞, say, which is what an over-cautious reader might choose for "there is no self-loop" — the collapse would fail, the recurrence would keep its two-part shape, and the connection to matrix multiplication would be invisible. The definition in Equation (15.1) was chosen to make Equation (15.5) true.

Let us check Equation (15.5) on one entry of the running example before we trust it. Take $i = 2$, $j = 4$, $m = 2$. With $L^{(1)} = W$, the five candidates are

$$\begin{aligned}
k = 1 &: l_{21}^{(1)} + w_{14} = \infty + \infty = \infty, \\
k = 2 &: l_{22}^{(1)} + w_{24} = 0 + 1 = 1, \\
k = 3 &: l_{23}^{(1)} + w_{34} = \infty + \infty = \infty, \\
k = 4 &: l_{24}^{(1)} + w_{44} = 1 + 0 = 1, \\
k = 5 &: l_{25}^{(1)} + w_{54} = 7 + 6 = 13,
\end{aligned}$$

so $l_{24}^{(2)} = \min\{\infty, 1, \infty, 1, 13\} = 1$. Two candidates tie at 1: the genuine one-edge route $2 \to 4$ counted through $k = 2$ (zero edges to reach 2, then the edge $2 \to 4$), and the same route counted through $k = 4$ (one edge to reach 4, then the zero-weight "stay put"). That duplication is the price of the simplification, and it costs nothing.

15.4.6 When does the sequence stop changing?

We now know how to get from $L^{(m-1)}$ to $L^{(m)}$. How far must we go? The argument has two halves.

First half: every shortest path has at most $n - 1$ edges. This is Theorem 13.4 of Chapter 13; the argument is short enough to repeat. Suppose $\delta(i, j) < \infty$, so some path from i to j exists. Claim: there is a shortest path from i to j that is **simple**, meaning it repeats no vertex. For suppose a shortest path p repeated a vertex v. Then p contains a cycle through v. That cycle has weight ≥ 0, because the graph has no negative-weight cycle. Deleting it therefore produces a path from i to j of weight no greater than $w(p)$, and with strictly fewer edges. Repeat until no vertex repeats. What remains is a simple path, still of minimum weight. A simple path visits at most n distinct vertices and therefore uses at most $n - 1$ edges.

Second half: extra edges never help. A path from i to j with more than $n - 1$ edges cannot have lower weight than a shortest path from i to j — by definition of "shortest", nothing can. So raising the budget beyond $n - 1$ can never lower the answer.

Put the halves together. At budget $n - 1$ we can already afford the best simple path, and beyond $n - 1$ nothing improves. Therefore

$$\delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots \tag{15.6}$$

for every pair i, j. The sequence $l_{ij}^{(0)}, l_{ij}^{(1)}, l_{ij}^{(2)}, \ldots$ is non-increasing, and it reaches its limit no later than step $n-1$ and stays there for ever. This is the termination condition for everything in §15.4 and §15.5 (it is equation (25.3) in the third edition of Cormen et al.).

> **WHERE THE NEGATIVE-CYCLE ASSUMPTION IS USED.** In the first half, and only there. If a negative-weight cycle existed on a route from i to j, deleting it would *raise* the weight, so we could not delete it, simple paths would not be optimal, and $l_{ij}^{(m)}$ would keep falling for ever — correctly, because $\delta(i, j)$ would be $-\infty$. Equation (15.6) is the precise point at which "no negative cycles" earns its keep.

15.4.7 Step 3: computing bottom up — EXTEND-SHORTEST-PATHS

The plan is this: from the input matrix $W = (w_{ij})$, compute the series of matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, where $L^{(m)} = (l_{ij}^{(m)})$, and the last matrix $L^{(n-1)}$ contains the actual shortest-path weights. The series starts with

$$l_{ij}^{(1)} = w_{ij} \quad \text{for all } i, j \in V, \qquad \text{so } L^{(1)} = W.$$

Why is $L^{(1)}$ equal to W rather than to something computed? Apply Equation (15.5) with $m = 1$: $l_{ij}^{(1)} = \min_k \left\{ l_{ik}^{(0)} + w_{kj} \right\}$. But $l_{ik}^{(0)}$ is 0 when $k = i$ and ∞ otherwise, so only the candidate $k = i$ survives, giving $l_{ij}^{(1)} = 0 + w_{ij} = w_{ij}$. The best route using at most one edge is the direct edge if there is one, and nothing if there is not. Exactly as it should be.

The step from one matrix to the next is a procedure called EXTEND-SHORTEST-PATHS, shown in **Figure 15.7**.

```
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.

Read it as English. Line 1 asks the matrix how many rows it has. Line 2 makes room for the answer $L' = (l'_{ij})$. Lines 3 and 4 walk over every cell of the answer. Line 5 sets that cell to ∞ — the identity for min, the value that loses against everything. Line 6 sweeps the intermediate vertex k from 1 to n, and line 7 keeps the running minimum:

$$l'_{ij} = \min(l'_{ij},\ l_{ik} + w_{kj}).$$

Line 8 returns L'. Three nested loops of n iterations each, with $O(1)$ work inside, give a running time of $\Theta(n^3)$.

15.4.8 The analogy with ordinary matrix multiplication

Now comes the observation that gives the section its name. Put EXTEND-SHORTEST-PATHS beside the textbook procedure for multiplying two square matrices, shown in **Figure 15.8**.

```
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.

They are the same procedure. Line for line:

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. Everything else — the loop structure, the index pattern, the running time — is identical.

Line	SQUARE-MATRIX-MULTIPLY	EXTEND-SHORTEST-PATHS
1	$n = A.rows$	$n = L.rows$
2	let C be a new $n \times n$ matrix	let $L' = (l'_{ij})$ be a new $n \times n$ matrix
3–4	**for** i, **for** j	**for** i, **for** j
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'

As **Table 15.2** sets out line by line, the substitution is:

$$+ \ \rightarrow\ \min, \qquad \times \rightarrow\ +, \qquad 0\ \rightarrow\ \infty.$$

That is all. The ordinary product accumulates $c_{ij} = \sum_k a_{ik}\, b_{kj}$; ours accumulates $l'_{ij} = \min_k\{l_{ik} + w_{kj}\}$. Both sweep k across row i of the left matrix and column j of the right matrix, combining the k-th pair and folding the results together.

Because it *is* a product, we may write it as one. Let $A \odot B$ denote the matrix returned by EXTEND-SHORTEST-PATHS(A, B). Then the whole plan becomes a sequence of powers:

$$\begin{aligned} L^{(1)} &= L^{(0)} \odot W = W, \\ L^{(2)} &= L^{(1)} \odot W = W^2, \\ L^{(3)} &= L^{(2)} \odot W = W^3, \\ &\vdots \\ L^{(n-1)} &= L^{(n-2)} \odot W = W^{n-1}, \end{aligned}$$

and by Equation (15.6) the matrix $L^{(n-1)} = W^{n-1}$ contains the shortest-path weights. The superscripts on W here are genuine powers — repeated $\odot$-multiplication — while the parenthesized superscripts on L are edge budgets. The two agree in value, which is exactly why the notation is worth the risk of confusion.

15.4.9 The $(\min, +)$ semiring, and why the analogy is exact

The resemblance is not a coincidence or a mnemonic. It is an instance of a general algebraic fact, and knowing it turns one algorithm into a family.

The schoolbook matrix product works over any **semiring**: a set with two operations, an "addition" $\oplus$ that is associative and commutative with an identity $\bar{0}$, a "multiplication" $\otimes$ that is associative with an identity $\bar{1}$, and a distributive law $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$. Real numbers with $+$ and $\times$ form one. So do the numbers $\mathbb{R} \cup \{\infty\}$ with $\oplus = \min$ and $\otimes = +$ — the $(\min, +)$ **semiring**, often called the *tropical* semiring:

- min is associative and commutative, and its identity is $\bar{0} = \infty$, because $\min(x, \infty) = x$;
- $+$ is associative, and its identity is $\bar{1} = 0$, because $x + 0 = x$;
- distributivity holds: $a + \min(b, c) = \min(a + b, a + c)$, which is true for real numbers and is exactly the statement that adding a constant to two journeys does not change which is shorter.

Three consequences follow immediately and for free.

The identity matrix. In the $(\min, +)$ semiring the identity matrix has $\bar{1} = 0$ on the diagonal and $\bar{0} = \infty$ off it — which is precisely the matrix $L^{(0)}$ of Equation (15.3). So $L^{(0)} \odot A = A$ for every A, and our base case is not an arbitrary choice but the only possible one.

Associativity. Matrix multiplication over a semiring is associative, so $(A \odot B) \odot C = A \odot (B \odot C)$. §15.5 is entirely built on this.

Other problems, free of charge. Swap in a different semiring and the same triple loop solves a different problem. With $\oplus =$ OR and $\otimes =$ AND you get reachability — that is §15.7. With $\oplus = \max$ and $\otimes = \min$ you get widest-path (maximum bottleneck). With $\oplus = +$ and $\otimes = \times$ over path counts you get the number of shortest paths. Exercise 15.14 and its solution demonstrate all three on real data.

15.4.10 SLOW-ALL-PAIRS-SHORTEST-PATHS

Wrapping the repeated product in a loop gives the procedure of **Figure 15.9**.

```
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 $\Theta(n^4)$.

Line 2 sets $L^{(1)} = W$, as §15.4.7 justified. Line 3 runs m from 2 to $n-1$, and line 5 computes $L^{(m)}$ from $L^{(m-1)}$ by one product with W. Line 6 returns $L^{(n-1)}$, which by Equation (15.6) holds the shortest-path weights.

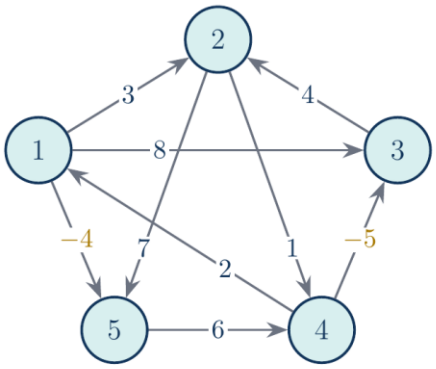
The running time is $n-2$ products, each $\Theta(n^3)$, hence

$$\Theta((n-2)\cdot n^3) = \Theta(n^4).$$

It is called SLOW for a reason: on a dense graph this is no better than $|V| \times$ Bellman–Ford from **Table 15.1**. Its purpose is not speed but *insight* — and §15.5 will turn that insight into a real saving.

15.4.11 The running example, every intermediate matrix

Figure 15.10 shows the running example together with every matrix SLOW-ALL-PAIRS-SHORTEST-PATHS produces on it.



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Figure 15.10 The five-vertex graph (Figure 25.1 in the third edition of Cormen et al.) 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 $L^{(4)}$.

The program of **Listing 15.2** reproduces them all. Every matrix printed in the rest of this chapter is output that a program on the page actually produced; none has been typed in by hand.

Listing 15.2 SLOW-ALL-PAIRS-SHORTEST-PATHS in full, printing every intermediate matrix. Compile with `g++ -std=c++17 -O2 -o slow_apsp slow_apsp.cpp`.

```
// Chapter 15, Listing: SLOW-ALL-PAIRS-SHORTEST-PATHS on the running example graph.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

const long long INF = 1000000000LL;      // stands for infinity; never added to itself

typedef vector<vector<long long>> Matrix;

// EXTEND-SHORTEST-PATHS(L, W): the "matrix product" L . W in the (min, +) semiring.
// Ordinary multiplication would write c_ij = c_ij + a_ik * b_kj; here we replace
// the sum by a minimum and the product by a sum.
Matrix extendShortestPaths(const Matrix& L, const Matrix& W) {
    int n = (int)L.size();
    Matrix Lp(n, vector<long long>(n, INF));      // line 2: a new n x n matrix L'
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            Lp[i][j] = INF;                       // line 5: l'_ij = infinity
            for (int k = 0; k < n; ++k)
                if (L[i][k] < INF && W[k][j] < INF)          // guard against INF + INF
                    Lp[i][j] = min(Lp[i][j], L[i][k] + W[k][j]);   // line 7
        }
    return Lp;
}

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 15.2 (Continued) *SLOW-ALL-PAIRS-SHORTEST-PATHS in full, printing every intermediate matrix. Compile with g++ -std=c++17 -O2 -o slow_apsp slow_apsp.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}
    };

    printMatrix("W", W);

    // SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
    Matrix L = W;                       // line 2: L^(1) = W
    printMatrix("L^(1)", L);
    for (int m = 2; m <= n - 1; ++m) {  // line 3: for m = 2 to n-1
        L = extendShortestPaths(L, W);  // line 5
        printMatrix("L^(" + to_string(m) + ")", L);
    }

    // One extra product, to see that nothing further changes.
    Matrix L5 = extendShortestPaths(L, W);
    printMatrix("L^(5)", L5);
    cout << "L^(5) equals L^(4)? " << (L5 == L ? "yes" : "no") << '\n';
    return 0;
}
```

Its output, with the `W` block omitted because it repeats `L^(1)`:

```
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^(3) =
       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

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

L^(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

L^(5) equals L^(4)? yes
```

Compare these matrices with the ones in **Figure 15.10**: they agree entry for entry. And the last line confirms Equation (15.6) empirically — the sequence has stopped moving by $m = n - 1 = 4$, and a fifth product changes nothing.

15.4.12 One matrix computed cell by cell

The output above is easy to check but hard to *feel*. Let us compute one whole row of $L^{(2)}$ by hand, so that nothing remains mysterious. We use Equation (15.5) with $m = 2$, and $L^{(1)} = W$ throughout.

Row 4 of $L^{(2)}$. For each destination j we minimize $l_{4k}^{(1)} + w_{kj}$ over $k = 1,2,3,4,5$. Row 4 of W is $(2,\ \infty,\ -5,\ 0,\ \infty)$, so only $k \in \{1,3,4\}$ can contribute a finite term.

$l_{41}^{(2)}$: the candidates are $l_{41}^{(1)} + w_{11} = 2 + 0 = 2$; $l_{42}^{(1)} + w_{21} = \infty + \infty = \infty$; $l_{43}^{(1)} + w_{31} = -5 + \infty = \infty$; $l_{44}^{(1)} + w_{41} = 0 + 2 = 2$; $l_{45}^{(1)} + w_{51} = \infty + \infty = \infty$. Minimum: **2**. That is the direct edge $4 \to 1$.

$l_{42}^{(2)}$: candidates $2 + 3 = 5$ (go $4 \to 1$ then $1 \to 2$); $\infty + 0 = \infty$; $-5 + 4 = \mathbf{-1}$ (go $4 \to 3$ then $3 \to 2$); $0 + \infty = \infty$; $\infty + \infty = \infty$. Minimum: $\mathbf{-1}$. Here is where the negative edge earns its keep — the two-edge route through vertex 3 costs -1, and there was no one-edge route at all.

$l_{43}^{(2)}$: candidates $2 + 8 = 10$; $\infty + \infty = \infty$; $-5 + 0 = -5$; $0 + (-5) = -5$; $\infty + \infty = \infty$. Minimum: $\mathbf{-5}$, the direct edge, counted twice as in §15.4.5.

$l_{44}^{(2)}$: candidates $2+\infty=\infty$; $\infty+1=\infty$; $-5+\infty=\infty$; $0+0=0$; $\infty+6=\infty$. Minimum: $\mathbf{0}$. Staying put is still free — and note that no two-edge cycle through vertex 4 is negative, which is the local evidence that this graph has no negative-weight cycle.

$l_{45}^{(2)}$: candidates $2+(-4)=\mathbf{-2}$ (go $4\to1$ then $1\to5$); $\infty+7=\infty$; $-5+\infty=\infty$; $0+\infty=\infty$; $\infty+0=\infty$. Minimum: $\mathbf{-2}$.

So row 4 of $L^{(2)}$ is $(2,\ -1,\ -5,\ 0,\ -2)$ — exactly the fourth row printed above, and exactly the fourth row of $L^{(2)}$ in **Figure 15.10**.

Row 2 of $L^{(2)}$, computed the same way from row 2 of W, which is $(\infty, 0, \infty, 1, 7)$:

- $l_{21}^{(2)}=\min\{\infty+0,\ 0+\infty,\ \infty+\infty,\ 1+2,\ 7+\infty\}=\mathbf{3}$, via $2\to4\to1$.
- $l_{22}^{(2)}=\min\{\infty+3,\ 0+0,\ \infty+4,\ 1+\infty,\ 7+\infty\}=\mathbf{0}$.
- $l_{23}^{(2)}=\min\{\infty+8,\ 0+\infty,\ \infty+0,\ 1+(-5),\ 7+\infty\}=\mathbf{-4}$, via $2\to4\to3$.
- $l_{24}^{(2)}=\min\{\infty+\infty,\ 0+1,\ \infty+\infty,\ 1+0,\ 7+6\}=\mathbf{1}$ (computed already in §15.4.5).
- $l_{25}^{(2)}=\min\{\infty+(-4),\ 0+7,\ \infty+\infty,\ 1+\infty,\ 7+0\}=\mathbf{7}$, the direct edge.

Row 2 is therefore $(3,\ 0,\ -4,\ 1,\ 7)$: again exactly what the program printed. Notice how row 2 changed between $L^{(1)}$ and $L^{(2)}$: the two ∞ entries in columns 1 and 3 became 3 and -4, because with a budget of two edges vertex 2 can now reach vertices 1 and 3 by hopping through vertex 4. That is the whole algorithm in one sentence: *each product lets every route grow by one more edge.*

15.4.13 Where the matrices came from, in one picture

It is worth pausing to see what the successive matrices *mean*, because the meaning is what makes the next section possible.

$L^{(1)}$ is the map of one-hop journeys. $L^{(2)}$ is the map of journeys of at most two hops. $L^{(3)}$, at most three. $L^{(4)}$, at most four — and since a simple path in a five-vertex graph has at most four edges, $L^{(4)}$ is the answer. Read off the entry l_{15}: it is -4 in every matrix, because the direct edge $1\to5$ of weight -4 is already optimal and nothing beats it. Read off l_{12}: it is 3 in $L^{(1)}$, $L^{(2)}$ and $L^{(3)}$, then drops to 1 in $L^{(4)}$, because the cheapest route $1\to5\to4\to3\to2$ with weight $-4+6-5+4=1$ needs all four edges and cannot appear before the budget reaches four. That single entry is the reason the algorithm must run all the way to $m=n-1$ and cannot stop early on a hunch.

15.5 Improving the Running Time: Repeated Squaring

15.5.1 We only want the last matrix

One observation makes everything cheaper: we are **not** interested in computing *all* the $L^{(m)}$ matrices. We are interested only in $L^{(n-1)}$. The intermediate matrices were a means, not an end, and if there is a way to reach the last one without visiting all its predecessors, we should take it.

Without negative-weight cycles, Equation (15.6) says $L^{(m)}=L^{(n-1)}$ for **all** integers $m\geq n-1$. That is a stronger statement than "the sequence converges": it says the sequence is *exactly constant* from index $n-1$ onward. So it does not matter whether we compute $L^{(n-1)}$ or $L^{(n)}$ or $L^{(1000)}$ — as long as the index is at least $n-1$, we get the same matrix, and that matrix is the answer. We are free to overshoot.

15.5.2 Associativity is what we sell

The second ingredient is the associativity of the $\odot$ product, guaranteed by §15.4.9. Associativity says that the order in which we bracket a chain of products does not change the result. SLOW-ALL-PAIRS-SHORTEST-PATHS computes

$$W^4 = ((W \odot W) \odot W) \odot W$$

in three products. But associativity permits

$$W^4 = (W \odot W) \odot (W \odot W),$$

and the two inner products are *the same computation*, so we need only perform it once and reuse it. Two products instead of three. On a larger example the saving is dramatic: W^{1024} takes 1023 products the slow way and ten the clever way.

15.5.3 The sequence of squares

Figure 15.11 writes the sequence out.

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:

$$\begin{aligned}
L^{(1)} &= W, \\
L^{(2)} &= W^2 = W \cdot W, \\
L^{(4)} &= W^4 = W^2 \cdot W^2, \\
L^{(8)} &= W^8 = W^4 \cdot W^4, \\
&\vdots \\
L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}.
\end{aligned}$$

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 $L^{(2^{\lceil \lg(n-1) \rceil})}$ equals $L^{(n-1)}$ by Equation (15.6).

The sequence is

$$\begin{aligned}
L^{(1)} &= W, \\
L^{(2)} &= W^2 = W \odot W, \\
L^{(4)} &= W^4 = W^2 \odot W^2, \\
L^{(8)} &= W^8 = W^4 \odot W^4, \\
&\vdots \\
L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil - 1}} \odot W^{2^{\lceil \lg(n-1) \rceil - 1}}.
\end{aligned}$$

Every line after the first squares the line above it, so the edge budget doubles at every step: 1,2,4,8,16, … This is exactly the technique used in Chapter 3 to compute a^b in $\lg b$ multiplications, applied to matrices in a different semiring.

The symbol $\lceil x \rceil$ is the **ceiling** of x: the smallest integer that is at least x. So $\lceil 3.2 \rceil = 4$, $\lceil 4 \rceil = 4$, $\lceil \lg 4 \rceil = 2$. And lg means $\log_2$, as it does everywhere in this book.

15.5.4 Why $\lceil \lg(n-1) \rceil$ products are enough

The argument is two lines. Here it is with every step shown.

We stop when the exponent first reaches or passes $n-1$. After t squarings the exponent is 2^t. We therefore want the smallest t with $2^t \geq n-1$. Taking lg of both sides — legitimate, because lg is increasing — gives $t \geq \lg(n-1)$, and since t must be an integer, the smallest such t is

$$t = \lceil \lg(n-1) \rceil.$$

At that point the exponent is $2^{\lceil \lg(n-1) \rceil}$, which satisfies

$$2^{\lceil \lg(n-1) \rceil} \geq n-1$$

by construction. By Equation (15.6), any index at or beyond $n-1$ gives the same matrix, so

$$L^{(2^{\lceil \lg(n-1) \rceil})} = L^{(n-1)},$$

and the last product is the answer. In one sentence: *since* $2^{\lceil \lg(n-1) \rceil} \geq n-1$, *the last product* $L^{(2^{\lceil \lg(n-1) \rceil})}$ *is equal to* $L^{(n-1)}$.

Worth dwelling on: we usually compute a matrix with a *larger* edge budget than we need, and that is fine. Overshooting costs nothing because the sequence is constant past $n-1$. If a colleague objects that W^{16} "counts paths with sixteen edges" in a graph where no useful path has more than nine, the answer is: yes, and those paths never win, because any path with a repeated vertex can have its cycle deleted without increasing its weight.

> **A CONCRETE COUNT.** For $n = 5$: $\lceil \lg 4 \rceil = 2$ products, against $n-2 = 3$ for the slow version. A saving of one. For $n = 1000$: $\lceil \lg 999 \rceil = 10$ products (since $2^9 = 512 < 999 \leq 1024 = 2^{10}$), against 998. A saving of a factor of one hundred. For $n = 10^6$: 20 products against 999,998 — a factor of fifty thousand. The bigger the graph, the more the trick is worth.

15.5.5 FASTER-ALL-PAIRS-SHORTEST-PATHS

The procedure is shown in **Figure 15.12**.

```
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**.

Compare it line by line with SLOW-ALL-PAIRS-SHORTEST-PATHS and you will find exactly two changes. First, line 6 calls EXTEND-SHORTEST-PATHS($L^{(m)}, L^{(m)}$) — squaring — where the slow version called EXTEND-SHORTEST-PATHS($L^{(m-1)}, W$). Second, line 7 doubles m where the slow version added one. The loop condition on line 4, `while m < n - 1`, is what makes the count come out at $\lceil \lg(n-1) \rceil$: the loop body runs as long as the current budget is short of $n-1$, and each body doubles it.

Because each of the $\lceil \lg(n-1) \rceil$ matrix products takes $\Theta(n^3)$ time, FASTER-ALL-PAIRS-SHORTEST-PATHS runs in

$$\Theta(n^3 \lceil \lg(n-1) \rceil) = \Theta(n^3 \lg n).$$

That is better than $\Theta(n^4)$ by a factor of $n/\lg n$ — for $n = 1000$, a factor of about a hundred.

15.5.6 The example again, in two products

Listing 15.3 runs the squaring version on the running example.

Listing 15.3 FASTER-ALL-PAIRS-SHORTEST-PATHS, the repeated-squaring version. Compile with `g++ -std=c++17 -O2 -o faster_apsp faster_apsp.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 15.3 (Continued) *FASTER-ALL-PAIRS-SHORTEST-PATHS, the repeated-squaring version. Compile with g++ -std=c++17 -O2 -o faster_apsp faster_apsp.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;
}
```

Its output:

```
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
```

Three things to notice. The final matrix $L^{(4)}$ is **identical** to the $L^{(4)}$ that the slow version produced in §15.4.11 — same answer, fewer products. The matrix $L^{(3)}$ is never computed at all; the algorithm leaps from budget 2 to budget 4. And the count at the bottom confirms the theory: two products, exactly $\lceil \lg 4 \rceil$.

Table 15.3 lays the two schedules side by side.

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	$n - 2 = 3$		$\lceil \lg(n-1) \rceil = 2$	

15.5.7 What repeated squaring does *not* give us

Two honest caveats, because a textbook that only reports good news teaches badly.

It is still $\Theta(n^3 \lg n)$, which Floyd–Warshall beats. The next section presents an algorithm that solves the same problem in $\Theta(n^3)$ — no logarithm at all — with a smaller constant factor and less memory. If all you want is the answer, use Floyd–Warshall. The value of §15.4 and §15.5 is the *idea*: the observation that shortest paths live in a semiring, which is what generalizes to transitive closure in §15.7, to widest paths, to path counting, and to the fast matrix-multiplication algorithms that lie beyond the scope of this book.

Strassen does not help here. A natural thought is: if EXTEND-SHORTEST-PATHS is matrix multiplication, why not use a sub-cubic matrix multiplication algorithm and get all-pairs shortest paths in $O(n^{2.81} \lg n)$? The answer is that Strassen's algorithm and its descendants rely on **subtraction** — they compute sums and differences of submatrices and cancel terms. The $(\min, +)$ semiring has no subtraction: there is no operation that undoes min. So the fast multiplication algorithms simply do not apply. This is not a gap waiting to be filled by a cleverer student; it is a genuine structural obstacle, and the sub-cubic all-pairs algorithms that do exist — Zwick's, for instance [8] — take a quite different route.

15.6 The Floyd–Warshall Algorithm

Three facts introduce it: it takes a weighted directed graph $G = (V, E)$; negative-weight edges are allowed but negative-weight cycles are not; and it is a dynamic-programming algorithm running in $\Theta(V^3)$ time. It is the algorithm most working programmers reach for when they need all-pairs shortest paths, and the reason is that its inner loop is five lines long, has no data structures in it at all, and runs at the speed of memory.

15.6.1 A different way to cut a path

Everything in §15.4 hung on one choice: we characterized a shortest path by **how many edges** it used, and built up the answer by raising that budget. Floyd and Warshall make a different choice. They characterize a shortest path by **which vertices it is allowed to pass through**, and build up the answer by enlarging that permitted set. The whole difference between the two algorithms is this one decision, and it is worth seeing how much follows from it.

First, a definition.

> **DEFINITION.** An **intermediate vertex** of a simple path $p = \langle v_1, v_2, \ldots, v_l \rangle$ is any vertex of p other than v_1 or v_l — that is, any vertex in the set $\{v_2, v_3, \ldots, v_{l-1}\}$.

The endpoints are not intermediate. A one-edge path $\langle i, j \rangle$ has no intermediate vertices at all. A two-edge path $\langle i, k, j \rangle$ has exactly one, namely k.

Now number the vertices $V = \{1,2, \dots, n\}$ and fix some k between 0 and n. Consider the set $\{1,2, \dots, k\}$ — the first k vertices in the numbering. For a pair i, j, look at **all paths from i to j whose intermediate vertices are all drawn from** $\{1,2, \dots, k\}$. The endpoints i and j need not be in that set; only what happens in between is restricted. Let p be a minimum-weight path among them. Observe that p is simple — a repeated vertex would give a cycle of weight ≥ 0, which can be deleted without raising the weight, exactly as in §15.4.6.

The algorithm now exploits a relationship between p and shortest paths from i to j with all intermediate vertices in the smaller set $\{1,2, \dots, k-1\}$.

15.6.2 The two cases

Figure 15.13 states the relationship.

Let p be a shortest path from i to j whose intermediate vertices all lie in $\{1, 2, \dots, k\}$. 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 $\{1, \dots, k-1\}$, and p is also a shortest path from i to j with all intermediate vertices in $\{1, \dots, k-1\}$.
- 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 $\{1, \dots, k-1\}$.

Figure 15.13 The structure of a shortest path whose intermediate vertices lie in $\{1, \dots, k\}$, split according to whether k itself is used.

The relationship depends on **whether or not k is an intermediate vertex of p**, and there are only these two possibilities.

Case 1: k is not an intermediate vertex of p. Then all intermediate vertices of p already lie in $\{1,2, \dots, k-1\}$. So a shortest path from i to j with intermediates in $\{1, \dots, k-1\}$ is *also* a shortest path from i to j with intermediates in $\{1, \dots, k\}$ — the extra permission was not used, and granting a permission you do not use changes nothing.

Case 2: k is an intermediate vertex of p. Then we may decompose p at k:

$$i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j,$$

where p_1 is the part of p before k and p_2 the part after. **Figure 15.14** draws exactly this.

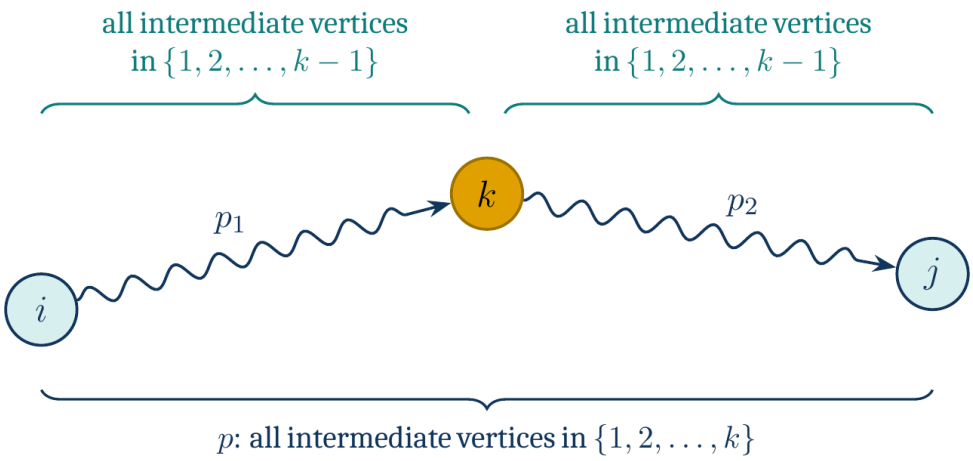


Figure 15.14 Path p (Figure 25.3 in the third edition of Cormen et al.) is a shortest path from i to j and k is its highest-numbered intermediate vertex. Both halves p_1 and p_2 then have all their intermediate vertices in $\{1,2, \ldots, k-1\}$.

15.6.3 Why both halves are restricted to $\{1, \ldots, k-1\}$

This is the step beginners most often skip. Read it slowly.

By Theorem 13.1, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1,2, \ldots, k\}$ — because it is a subpath of a shortest path, and its intermediates are among p's intermediates.

But now: **vertex k is not an intermediate vertex of path p_1.** Why not? Because k is the *endpoint* of p_1, and by the definition in §15.6.1 an endpoint is never intermediate. And k cannot appear a second time in the middle of p_1, because p is simple and so no vertex repeats.

Therefore all intermediate vertices of p_1 are in the set $\{1,2, \ldots, k-1\}$, and p_1 is a shortest path from i to k with all intermediate vertices in $\{1,2, \ldots, k-1\}$. Similarly, p_2 is a shortest path from k to vertex j with all intermediate vertices in $\{1,2, \ldots, k-1\}$.

> **IN PLAIN LANGUAGE.** Suppose you are told the cheapest route from Keputih to Darmo that is allowed to pass through any of the first seven vertices, and suppose it does pass through vertex 7. Cut it in half at vertex 7. Each half is a cheapest route in its own right (Theorem 13.1) — and neither half passes *through* vertex 7, because vertex 7 is now an endpoint, and the route never visits a place twice. So each half is a cheapest route that only uses the first **six** vertices in the middle. Two smaller problems of exactly the same shape. That is optimal substructure, and it is why dynamic programming works here.

15.6.4 The recurrence

We can now define the quantity and write the recurrence, summarized in **Figure 15.15**.

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.

DEFINITION. Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j **for which all intermediate vertices are in the set** $\{1,2, \ldots, k\}$.

As with $l_{ij}^{(m)}$, the parenthesized superscript is a label and not a power. But it now labels a different thing: (m) counted edges, (k) counts *permitted intermediate vertices*. The two families of matrices are different, they are computed differently, and only their last members agree.

The base case. When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has **no intermediate vertices at all** — the set $\{1, \ldots, 0\}$ is empty. Such a path has at most one edge, so it is either the direct edge (i, j), or nothing. Hence

$$d_{ij}^{(0)} = w_{ij},$$

which in matrix form says $D^{(0)} = W$.

The recursive case. For $k \geq 1$ the two cases of §15.6.2 become the two arguments of a minimum. Case 1 — do not use k — contributes $d_{ij}^{(k-1)}$. Case 2 — use k — contributes $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$, the best way to get from i to k using only low-numbered intermediates, plus the best way to get from k to j using only low-numbered intermediates. Exactly one of the two cases holds for the optimal path, and since we take a minimum over both, we cannot be wrong either way. So

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1. \end{cases} \tag{15.7}$$

Why the answer is $D^{(n)}$. For **any** path, all intermediate vertices are in the set $\{1,2, \ldots, n\}$ — there are no other vertices to be in. So the restriction at $k = n$ is no restriction at all, and

$$d_{ij}^{(n)} = \delta(i, j) \quad \text{for all } i, j \in V. \tag{15.8}$$

The matrix $D^{(n)} = (d_{ij}^{(n)})$ is the answer we want: it meets the goal Equation (15.2) set in §15.2.2.

A COMMON MISREADING. In the term $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$, both superscripts are $(k - 1)$, not (k). That is the whole content of §15.6.3: the two halves are restricted to the *smaller* set. Writing

> $d_{ik}^{(k)} + d_{kj}^{(k)}$ would be circular — it would define $D^{(k)}$ in terms of itself — and it would allow a route from i to k that passes through k, which is nonsense.

Compare Equation (15.7) with Equation (15.5) for a moment. The matrix-multiplication recurrence takes a minimum over n candidates and must be applied $n-1$ times; the Floyd–Warshall recurrence takes a minimum over just **two** candidates and must be applied n times. That is the source of the whole speed difference: n rounds of n^2 constant-time updates is $\Theta(n^3)$, where n rounds of n^2 minima-over-n would be $\Theta(n^4)$.

15.6.5 The algorithm

Figure 15.16 gives the procedure.

```
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.

Line 2 sets $D^{(0)} = W$, the base case. Line 3 runs k from 1 to n: this is the loop that enlarges the permitted set, one vertex at a time. Lines 5 and 6 walk over every pair (i, j). Line 7 applies the recurrence. Line 8 returns $D^{(n)}$, which by Equation (15.8) is the matrix of shortest-path weights.

The values $d_{ij}^{(k)}$ are computed **in order of increasing values of** k. The next subsection explains why that is not a stylistic preference but a correctness requirement.

15.6.6 Why k must be the outermost loop

Here is the question every reader should ask and few do: the body of the triple loop is symmetric-looking, so why can we not write the loops in whatever order we please?

The answer is the dependency structure. To compute $d_{ij}^{(k)}$ we need three values, all with superscript $(k-1)$: $d_{ij}^{(k-1)}$, $d_{ik}^{(k-1)}$ and $d_{kj}^{(k-1)}$. Every value we read must belong to the **previous** round. If k is the outermost loop, that is automatic: when we start round k, every entry of the matrix carries superscript $(k-1)$, and we finish the round before moving on. If k is *not* outermost — if, say, i is outermost and k innermost — then when we compute the entry d_{ij} we may reach for d_{kj} in a row k that this pass has not yet visited, and we get a value from an earlier, weaker stage of the computation than the recurrence requires.

The effect is not subtle. **Listing 15.4** builds two four-vertex graphs designed to expose it. Graph (a) is the directed path $1 \to 4 \to 3 \to 2$, each edge of weight 1: the route from 1 to 2 needs vertices

4 and 3 as intermediates, both numbered *higher* than the destination. Graph (b), which is also the answer to Exercise 15.9, has edges 1 → 4 of weight 6, 4 → 3 of weight 3, 4 → 2 of weight 6 and 3 → 2 of weight 1.

Listing 15.4 The same triple loop with k outermost and with k innermost, on two graphs built to tell them apart. Compile with `g++ -std=c++17 -O2 -o ex_loop_order ex_loop_order.cpp`.

```
// Chapter 15, Listing: what happens when k is not the outermost loop.
// Two graphs: on (a) the wrong order loses a path entirely, on (b) it finds a
// path but overcharges for it. Both have four vertices and non-negative weights.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

const long long INF = 1000000000LL;
typedef vector<vector<long long>> Matrix;

Matrix loopKOutermost(Matrix D) {
    int n = (int)D.size();
    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][j] = min(D[i][j], D[i][k] + D[k][j]);
    return D;
}

Matrix loopKInnermost(Matrix D) {
    int n = (int)D.size();
    for (int i = 0; i < n; ++i)                        // WRONG: i outermost
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                if (D[i][k] < INF && D[k][j] < INF)
                    D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
    return D;
}

void printMatrix(const string& name, const Matrix& M) {
    cout << name << '\n';
    for (size_t i = 0; i < M.size(); ++i) {
        cout << "   ";
        for (size_t j = 0; j < M.size(); ++j) {
            if (M[i][j] >= INF) cout << setw(6) << "inf";
            else                cout << setw(6) << M[i][j];
        }
        cout << '\n';
    }
}
```

Listing 15.4 (Continued) *The same triple loop with k outermost and with k innermost, on two graphs built to tell them apart. Compile with g++ -std=c++17 -O2 -o ex_loop_order ex_loop_order.cpp.*

```
void compare(const string& title, const Matrix& W, int si, int sj) {
    Matrix right = loopKOutermost(W);
    Matrix wrong = loopKInnermost(W);
    cout << title << '\n';
    printMatrix("  k outermost (correct):", right);
    printMatrix("  i outermost, k innermost (wrong):", wrong);
    cout << "  the two agree: " << (right == wrong ? "yes" : "NO") << '\n';
    cout << "  delta(" << si + 1 << "," << sj + 1 << "): correct = " << right[si][sj]
         << ", wrong = ";
    if (wrong[si][sj] >= INF) cout << "inf (no path found at all)\n";
    else                      cout << wrong[si][sj] << '\n';

    // Repeating the wrong loop order does eventually converge -- it is merely
    // slower and needs an extra correctness argument.
    Matrix repaired = W;
    int passes = 0;
    while (repaired != right && passes < 10) { repaired = loopKInnermost(repaired);
++passes; }
    cout << "  passes of the wrong order needed before it matches: " << passes << "\n\n";
}

int main() {
    // (a) The directed path 1 -> 4 -> 3 -> 2, each edge of weight 1. The
    // intermediates a shortest path from 1 to 2 needs are 4 and 3 -- both
    // numbered higher than the destination, which defeats the wrong order.
    Matrix a = {
        {  0, INF, INF,   1},
        {INF,   0, INF, INF},
        {INF,   1,   0, INF},
        {INF, INF,   1,   0}
    };
    // (b) Edges 1->4 (6), 4->3 (3), 4->2 (6), 3->2 (1). The wrong order finds
    // 1 -> 4 -> 2 at 12 and never discovers 1 -> 4 -> 3 -> 2 at 10.
    Matrix b = {
        {  0, INF, INF,   6},
        {INF,   0, INF, INF},
        {INF,   1,   0, INF},
        {INF,   6,   3,   0}
    };
    compare("(a) the wrong order loses the path entirely:", a, 0, 1);
    compare("(b) the wrong order finds a path, but overcharges:", b, 0, 1);
    return 0;
}
```

Its output:

```
(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
```

On graph (a) the wrong order does not merely give a slightly-too-large answer; it reports **no path at all** from vertex 1 to vertex 2, where in fact a path of weight 3 exists. Trace it. With i outermost, the very first thing the program does is fill in row 1 completely. When it reaches the cell (1,2) it sweeps k and looks for $d_{1k} + d_{k2}$. The useful combination is $k = 4$: $d_{14} = 1$ and d_{42} ought to be 2 — but row 4 has not been touched yet, so d_{42} is still ∞. The combination fails. By the time row 4 is filled in with $d_{42} = 2$, the program has long since moved past cell (1,2) and will never look at it again.

Graph (b) shows the quieter failure mode. The wrong order does find a route from 1 to 2 — the two-edge $1 \to 4 \to 2$ at cost $6 + 6 = 12$ — but it never discovers $1 \to 4 \to 3 \to 2$ at cost $6 + 3 + 1 = 10$, because when the j-loop reaches column 2 the entry d_{13} has not yet been improved from ∞ to 9; that happens one column later. A finite, plausible, wrong number is far harder to notice in production than an ∞.

The last line of each block is a genuinely useful piece of folklore: running the wrong loop order **repeatedly** does converge, here after two passes. But each pass costs $\Theta(n^3)$, and the number of passes needed depends on the vertex numbering, so the total cost is not bounded by $\Theta(n^3)$ and the algorithm has lost its guarantee. Get the loop order right.

> **THE SINGLE MOST COMMON FLOYD–WARSHALL BUG.** `for (i) for (j) for (k)` instead of `for (k) for (i) for (j)`. It compiles, it runs, it produces a matrix that looks plausible, and it is wrong. If you remember one thing from this section, remember that k goes first.

15.6.7 The predecessor matrix

The version in **Figure 15.16** returns distances only. To get routes we compute $\Pi^{(k)}$ alongside $D^{(k)}$, using the same case analysis.

DEFINITION. Let $\pi_{ij}^{(k)}$ be the predecessor of vertex j on a shortest path from i whose intermediate vertices all lie in $\{1,2,\ldots,k\}$.

The base case mirrors $d_{ij}^{(0)} = w_{ij}$. With no intermediates allowed, the only route from i to j is the direct edge, whose predecessor of j is i:

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty. \end{cases} \tag{15.9}$$

Starting from Equation (15.9), the recursive case follows the two cases of the recurrence. If the minimum in Equation (15.7) is achieved by the first argument — we did *not* route through k — then the route is unchanged and so is the predecessor. If the minimum is achieved by the second argument — we *did* route through k — then the new route ends with the tail half $k \rightsquigarrow j$, so the predecessor of j is whatever it was 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 second line is worth staring at, because the natural guess — $\pi_{ij}^{(k)} = k$ — is wrong. Vertex k is somewhere in the *middle* of the new route, not at its end. What we need is the vertex immediately before j, and on the segment $k \rightsquigarrow j$ that vertex is exactly $\pi_{kj}^{(k-1)}$. In code, the line is `P[i][j] = P[k][j];`, not `P[i][j] = k;`.

15.6.8 The running example, every $D^{(k)}$ and $\Pi^{(k)}$

Figure 15.17 shows the complete trace of Floyd–Warshall on the running example: six distance matrices and six predecessor matrices.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Figure 15.17 The complete sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by Floyd–Warshall for the graph of **Figure 15.2** (Figure 25.4 in the third edition of Cormen et al.). Row $k = 0$ is the input; row $k = 5$ is the answer.

Listing 15.5 is the full program that produces them, together with path reconstruction and negative-cycle detection.

Listing 15.5 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`.

```
// 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 15.5 (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 15.5 (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;
}
```

Its output, in full:

```
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
```

Every one of these twelve matrices agrees, entry for entry, with **Figure 15.17**. And $D^{(5)}$ agrees with the $L^{(4)}$ of §15.4.11: two completely different algorithms, one answer.

15.6.9 The round $k = 1$, cell by cell

Now we work one entire round by hand. Round $k = 1$ asks: *for every pair* (i, j)*, does routing through vertex 1 beat what we already have?* The candidate is $d_{i1}^{(0)} + d_{1j}^{(0)}$; the incumbent is $d_{ij}^{(0)}$; the winner is the smaller. All twenty-five cells, taken in the order the loops visit them, using $D^{(0)} = W$:

Row $i = 1$**.** Every candidate is $d_{11} + d_{1j} = 0 + d_{1j} = d_{1j}$, a tie with the incumbent. Nothing can change: $(0,3,8,\infty,-4)$ stays $(0,3,8,\infty,-4)$. In general, **row** k **and column** k **never change during round** k, because the candidate degenerates to the incumbent plus zero. That is a useful fact and §15.6.13 turns it into a theorem.

Row $i = 2$**.** Here $d_{21}^{(0)} = \infty$: vertex 2 has no route to vertex 1 using no intermediates, so every candidate $\infty + d_{1j}$ is ∞ and loses. Row 2 stays $(\infty, 0, \infty, 1,7)$. In general, **if** $d_{ik} = \infty$ **then row** i **cannot change during round** k — a test worth putting in the code, and **Listing 15.11** does.

Row $i = 3$**.** Same story: $d_{31}^{(0)} = \infty$, so row 3 stays $(\infty, 4,0, \infty, \infty)$.

Row $i = 4$**.** Now something happens, because $d_{41}^{(0)} = 2$ is finite — vertex 4 has a direct edge to vertex 1.

- $j = 1$: $\min(d_{41}^{(0)} = 2,\ d_{41}^{(0)} + d_{11}^{(0)} = 2 + 0 = 2) = 2$. Unchanged.
- $j = 2$: $\min(d_{42}^{(0)} = \infty,\ d_{41}^{(0)} + d_{12}^{(0)} = 2 + 3 = 5) = \mathbf{5}$. **Changed** from ∞ to 5: the route $4 \to 1 \to 2$ is the first route from 4 to 2 we have found. Since the improvement came from routing through $k = 1$, Equation (15.10) sets $\pi_{42}^{(1)} = \pi_{12}^{(0)} = 1$.

- $j = 3$: $\min(-5,\ 2 + 8 = 10) = -5$. Unchanged; the direct edge $4 \to 3$ of weight -5 is better than going round through 1.
- $j = 4$: $\min(0,\ 2 + \infty = \infty) = 0$. Unchanged.
- $j = 5$: $\min(d_{45}^{(0)} = \infty,\ 2 + (-4) = -2) = \mathbf{-2}$. **Changed**: the route $4 \to 1 \to 5$ costs $2 - 4 = -2$. And $\pi_{45}^{(1)} = \pi_{15}^{(0)} = 1$.

Row $i = 5$**.** $d_{51}^{(0)} = \infty$, so nothing changes: $(\infty, \infty, \infty, 6, 0)$.

Two cells changed in the whole round, both in row 4, and both are visible in the printed $D^{(1)}$ above: the 5 at position (4,2) and the -2 at position (4,5). The two corresponding entries of $\Pi^{(1)}$ both became 1. Everything else is untouched. That is one complete round of Floyd–Warshall, with no step omitted.

15.6.10 The round $k = 4$, where most of the work happens

Round $k = 1$ was quiet because vertex 1 has only one incoming edge. Round $k = 4$ is the loud one, because by then $D^{(3)}$ has plenty of finite entries and vertex 4 sits in the middle of the graph. Working from $D^{(3)}$, whose rows are $(0,3,8,4,-4)$, $(\infty, 0, \infty, 1, 7)$, $(\infty, 4, 0, 5, 11)$, $(2, -1, -5, 0, -2)$ and $(\infty, \infty, \infty, 6, 0)$:

Row 1 ($d_{14}^{(3)} = 4$): $j = 1$: $\min(0, 4 + 2 = 6) = 0$. $j = 2$: $\min(3, 4 + (-1) = 3) = 3$, a tie, so by Equation (15.10) the incumbent keeps the cell and π is unchanged. $j = 3$: $\min(8,\ 4 + (-5) = -1) = \mathbf{-1}$, **changed**, and $\pi_{13}^{(4)} = \pi_{43}^{(3)} = 4$. $j = 4$: $\min(4, 4 + 0) = 4$. $j = 5$: $\min(-4,\ 4 + (-2) = 2) = -4$.

Row 2 ($d_{24}^{(3)} = 1$): $j = 1$: $\min(\infty,\ 1 + 2 = 3) = \mathbf{3}$, **changed**, $\pi_{21}^{(4)} = \pi_{41}^{(3)} = 4$. $j = 2$: $\min(0,\ 1 + (-1) = 0) = 0$. $j = 3$: $\min(\infty,\ 1 + (-5) = -4) = \mathbf{-4}$, **changed**, $\pi_{23}^{(4)} = \pi_{43}^{(3)} = 4$. $j = 4$: $\min(1, 1 + 0) = 1$. $j = 5$: $\min(7,\ 1 + (-2) = -1) = \mathbf{-1}$, **changed**, $\pi_{25}^{(4)} = \pi_{45}^{(3)} = 1$.

Row 3 ($d_{34}^{(3)} = 5$): $j = 1$: $\min(\infty,\ 5 + 2 = 7) = \mathbf{7}$, **changed**, $\pi_{31}^{(4)} = 4$. $j = 2$: $\min(4,\ 5 + (-1) = 4) = 4$, a tie. $j = 3$: $\min(0,\ 5 + (-5) = 0) = 0$, a tie — and note this is the diagonal, which means the cycle $3 \to 2 \to 4 \to 3$ has weight exactly 0, not negative. $j = 4$: $\min(5, 5 + 0) = 5$. $j = 5$: $\min(11,\ 5 + (-2) = 3) = \mathbf{3}$, **changed**, $\pi_{35}^{(4)} = \pi_{45}^{(3)} = 1$.

Row 4 is row k, so nothing changes, exactly as §15.6.9 predicted.

Row 5 ($d_{54}^{(3)} = 6$): $j = 1$: $\min(\infty, 6 + 2 = 8) = \mathbf{8}$, **changed**, $\pi_{51}^{(4)} = 4$. $j = 2$: $\min(\infty, 6 + (-1) = 5) = \mathbf{5}$, **changed**, $\pi_{52}^{(4)} = \pi_{42}^{(3)} = 3$. $j = 3$: $\min(\infty, 6 + (-5) = 1) = \mathbf{1}$, **changed**, $\pi_{53}^{(4)} = 4$. $j = 4$: $\min(6, 6 + 0) = 6$. $j = 5$: $\min(0, 6 + (-2) = 4) = 0$.

Nine cells changed in round 4 — compare the printed $D^{(3)}$ and $D^{(4)}$ and you will count exactly nine differences. After this round every entry of D is finite: the graph is strongly connected, and vertex 4 was the hub that joined the pieces.

15.6.11 Reading routes out of Π

With $\Pi^{(5)}$ in hand, PRINT-ALL-PAIRS-SHORTEST-PATH of §15.2.3 turns any entry into a route. The remainder of **Listing 15.5**'s output does all twenty of them:

```
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
```

Take the first line and follow the recursion by hand. We want the route from 1 to 2. $\pi_{12}^{(5)} = 3$, so print the route from 1 to 3, then print "2". The route from 1 to 3: $\pi_{13}^{(5)} = 4$, so print the route from 1 to 4, then "3". The route from 1 to 4: $\pi_{14}^{(5)} = 5$, so print the route from 1 to 5, then "4". The route from 1 to 5: $\pi_{15}^{(5)} = 1$, so print the route from 1 to 1 — which is the base case, printing "1" — then "5". Unwinding: `1`, `5`, `4`, `3`, `2`. Check the weight: $w_{15} + w_{54} + w_{43} + w_{32} = -4 + 6 - 5 + 4 = 1$, which is $d_{12}^{(5)}$. Correct.

15.6.12 The running time

The running time is determined by the triply nested **for** loops of lines 3–7. Each execution of line 7 takes $O(1)$ time — one addition, one comparison, one assignment, no data structure — and line 7 executes exactly n^3 times. So the algorithm runs in time

$$\Theta(n^3).$$

The constant is small and the memory access pattern is nearly ideal: in round k, the inner j-loop walks along row i of D and along row k of D, both contiguous in memory, while d_{ik} is a single value held in a register. This is why Floyd–Warshall routinely outruns algorithms with better asymptotic bounds on graphs of a few thousand vertices.

The space is $\Theta(n^2)$ for D and another $\Theta(n^2)$ for Π. A literal reading of **Figure 15.16** would suggest $\Theta(n^3)$, because line 4 allocates a *new* matrix $D^{(k)}$ on every round, keeping all $n + 1$ of them. The next subsection shows that we need not.

15.6.13 Doing it in place, and why that is safe

Look again at **Listing 15.5**: it keeps one matrix `D` and overwrites it. That is what every real implementation does, and it requires proof, because the recurrence reads $d_{ik}^{(k-1)}$ and $d_{kj}^{(k-1)}$ while we are busy overwriting the matrix with round-k values. Could we read a value that has already been updated, and get the wrong answer?

THEOREM 15.1 (the in-place optimization is safe). Running the triple loop on a single matrix, overwriting d_{ij} in place, produces exactly the matrix $D^{(n)}$ that the version with $n+1$ separate matrices produces.

Proof. The only values read during round k that might have been altered during round k are those in row k and column k — that is, entries of the form d_{kj} or d_{ik}. So it suffices to show that **round k never changes row k or column k.**

Take an entry in row k, say d_{kj}. During round k its candidate replacement is $d_{kk} + d_{kj}$. Now d_{kk} is the weight of a cheapest route from k back to k; it is 0 if there is no cycle through k, and it is the weight of the cheapest such cycle otherwise — which is ≥ 0, since the graph has no negative-weight cycle. So $d_{kk} \geq 0$, hence $d_{kk} + d_{kj} \geq d_{kj}$, and the candidate never beats the incumbent. Row k is untouched. The argument for column k is identical: the candidate for d_{ik} is $d_{ik} + d_{kk} \geq d_{ik}$.

Therefore every value read during round k still carries its round-$(k-1)$ value at the moment it is read, whether or not we allocated a fresh matrix. The two versions compute the same thing. ■

Notice where the hypothesis went in: $d_{kk} \geq 0$, which is exactly "no negative-weight cycle". If the graph did contain one, the in-place version could differ from the textbook version — although in that case neither is computing anything meaningful, since the true answers are $-\infty$.

The saving is real: $\Theta(n^2)$ memory instead of $\Theta(n^3)$. At $n = 2000$ that is 32 megabytes instead of 64 gigabytes. It is the difference between an algorithm you can run and one you cannot.

15.6.14 Detecting negative-weight cycles for free

Everything so far assumed no negative-weight cycle. What if we are not sure? Floyd–Warshall answers the question at no extra cost, and the test is one line.

THE TEST. After the triple loop, examine the diagonal. The graph contains a negative-weight cycle **if and only if** $d_{ii} < 0$ for some vertex i.

Why it works. The entry d_{ii} is the weight of a cheapest closed walk from i back to i. If no negative cycle exists, no closed walk has negative weight, so $d_{ii} = 0$ (the empty walk) for every i. Conversely, if $d_{ii} < 0$ then some closed walk through i has negative weight, and a closed walk of negative weight must contain a negative cycle — decompose it into simple cycles; if all of them had weight ≥ 0 the total could not be negative. Finally, every negative cycle passes through some vertex, and by the time k has reached the highest-numbered vertex on that cycle, the whole cycle is available to the recurrence, so the corresponding diagonal entry has gone negative. **Listing 15.6** demonstrates all of this.

Listing 15.6 Negative-weight-cycle detection from the diagonal, on three graphs. Compile with `g++ -std=c++17 -O2 -o negcycle negcycle.cpp`.

```
// Chapter 15, Listing: detecting a negative-weight cycle from the diagonal of D.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

const long long INF = 1000000000LL;
typedef vector<vector<long long>> Matrix;

// Returns true when G contains a negative-weight cycle. D is left holding the
// result of the triple loop, whose diagonal is the evidence.
bool floydWarshallDetect(Matrix& D) {
    int n = (int)D.size();
    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];
    for (int i = 0; i < n; ++i) if (D[i][i] < 0) return true;
    return false;
}

void report(const string& title, Matrix W) {
    cout << title << '\n';
    bool bad = floydWarshallDetect(W);
    cout << "   diagonal after the triple loop: ";
    for (size_t i = 0; i < W.size(); ++i) cout << setw(4) << W[i][i];
    cout << "\n   negative-weight cycle? " << (bad ? "YES" : "no") << "\n\n";
}

int main() {
    // (a) The running example -- no negative cycle, despite two negative edges.
    report("(a) the running example (edges of weight -4 and -5, but no negative cycle):", {
        {  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}
    });
```

Listing 15.6 (Continued) *Negative-weight-cycle detection from the diagonal, on three graphs. Compile with g++ -std=c++17 -O2 -o negcycle negcycle.cpp.*

```
    // (b) A three-vertex graph with the cycle 1 -> 2 -> 3 -> 1 of total weight -1.
    report("(b) a graph whose cycle 1 -> 2 -> 3 -> 1 has weight 2 + 3 + (-6) = -1:", {
        {  0,   2, INF},
        {INF,   0,   3},
        { -6, INF,   0}
    });

    // (c) A two-vertex graph with a negative two-cycle.
    report("(c) a graph whose cycle 1 -> 2 -> 1 has weight 1 + (-4) = -3:", {
        {  0,   1},
        { -4,   0}
    });
    return 0;
}
```

Its output:

```
(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
```

Graph (a) is the running example: two of its nine edges are negative, and yet the diagonal is all zeros, which is the machine-checked proof that every result in §15.6.8 is meaningful. In (b) the cycle $1 \to 2 \to 3 \to 1$ has weight $2 + 3 - 6 = -1$, and all three diagonal entries go negative. In (c) the two-cycle has weight -3, and the diagonal reports -3 and -6 — note that the numbers off the diagonal in such a graph are meaningless, because the true answers are $-\infty$; only the *sign test* is trustworthy.

15.6.15 All four methods agree

Before moving on, it is worth checking the whole of §15.3 to §15.6 against itself. **Listing 15.7** runs $|V| \times$ Bellman–Ford, SLOW-APSP, repeated squaring, Floyd–Warshall and (looking ahead) Johnson on the running example and compares the five matrices.

Listing 15.7 Five algorithms, one graph, one answer. Compile with `g++ -std=c++17 -O2 -o crosscheck crosscheck.cpp`.

```
// Chapter 15, Listing: five routes to the same answer. Runs |V| x Bellman-Ford,
// SLOW-APSP, repeated squaring, Floyd-Warshall and Johnson on the running example
// and checks that every pair of methods produces exactly the same matrix.
#include <iostream>
#include <iomanip>
#include <vector>
#include <queue>
#include <string>
using namespace std;

const long long INF = 1000000000LL;
typedef vector<vector<long long>> Matrix;
struct Edge { int u, v; long long w; };

static const int n = 5;
static 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}
};
static 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}
};

Matrix extendShortestPaths(const Matrix& A, const Matrix& B) {
    Matrix C(n, vector<long long>(n, INF));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            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;
}

Matrix methodSlow() {
    Matrix L = W;
    for (int m = 2; m <= n - 1; ++m) L = extendShortestPaths(L, W);
    return L;
}
```

Listing 15.7 (Continued) *Five algorithms, one graph, one answer. Compile with g++ -std=c++17 -O2 -o crosscheck crosscheck.cpp.*

```
Matrix methodSquaring() {
    Matrix L = W;
    for (int m = 1; m < n - 1; m *= 2) L = extendShortestPaths(L, L);
    return L;
}

Matrix methodFloydWarshall() {
    Matrix D = W;
    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][j] = min(D[i][j], D[i][k] + D[k][j]);
    return D;
}

Matrix methodRepeatedBellmanFord() {
    Matrix D(n, vector<long long>(n, INF));
    for (int s = 0; s < n; ++s) {
        vector<long long> d(n, INF);
        d[s] = 0;
        for (int pass = 1; pass <= n - 1; ++pass)
            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;
        D[s] = d;
    }
    return D;
}

Matrix methodJohnson() {
    int s = n;
    vector<Edge> Ep = E;
    for (int v = 0; v < n; ++v) Ep.push_back({s, v, 0});
    vector<long long> h(n + 1, INF);
    h[s] = 0;
    for (int pass = 1; pass <= n; ++pass)
        for (const Edge& e : Ep)
            if (h[e.u] < INF && h[e.u] + e.w < h[e.v]) h[e.v] = h[e.u] + e.w;

    vector<vector<pair<int,long long>>> adj(n);
    for (const Edge& e : E) adj[e.u].push_back({e.v, e.w + h[e.u] - h[e.v]});
```

Listing 15.7 (Continued) *Five algorithms, one graph, one answer. Compile with g++ -std=c++17 -O2 -o crosscheck crosscheck.cpp.*

```
    Matrix D(n, vector<long long>(n, INF));
    for (int u = 0; u < n; ++u) {
        vector<long long> d(n, INF);
        d[u] = 0;
        priority_queue<pair<long long,int>, vector<pair<long long,int>>, greater<>> Q;
        Q.push({0, u});
        while (!Q.empty()) {
            auto [du, x] = Q.top(); Q.pop();
            if (du > d[x]) continue;
            for (auto [y, wt] : adj[x])
                if (d[x] + wt < d[y]) { d[y] = d[x] + wt; Q.push({d[y], y}); }
        }
        for (int v = 0; v < n; ++v) if (d[v] < INF) D[u][v] = d[v] + h[v] - h[u];
    }
    return D;
}

void printMatrix(const string& name, const Matrix& M) {
    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(6) << "inf";
            else                cout << setw(6) << M[i][j];
        }
        cout << '\n';
    }
}

int main() {
    vector<pair<string, Matrix>> results = {
        {"|V| x Bellman-Ford", methodRepeatedBellmanFord()},
        {"SLOW-APSP",          methodSlow()},
        {"repeated squaring",  methodSquaring()},
        {"Floyd-Warshall",     methodFloydWarshall()},
        {"Johnson",            methodJohnson()}
    };
    printMatrix("the agreed answer", results[0].second);
    cout << '\n';
    bool allAgree = true;
    for (size_t t = 1; t < results.size(); ++t) {
        bool same = (results[t].second == results[0].second);
        allAgree = allAgree && same;
        cout << setw(20) << left << results[t].first
             << " agrees with " << results[0].first << ": "
             << (same ? "YES" : "NO") << '\n';
```

Listing 15.7 (Continued) *Five algorithms, one graph, one answer. Compile with g++ -std=c++17 -O2 -o crosscheck crosscheck.cpp.*

```
    }
    cout << "\nall five methods agree: " << (allAgree ? "YES" : "NO") << '\n';
    return 0;
}
```

Its output:

```
the agreed answer:
        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

SLOW-APSP            agrees with |V| x Bellman-Ford: YES
repeated squaring    agrees with |V| x Bellman-Ford: YES
Floyd-Warshall       agrees with |V| x Bellman-Ford: YES
Johnson              agrees with |V| x Bellman-Ford: YES

all five methods agree: YES
```

That is the kind of check worth building into your own work. Five independent implementations of the same specification, cross-validated against each other, is far stronger evidence of correctness than any one of them tested alone.

15.7 Transitive Closure of a Directed Graph

15.7.1 The question, and the lazy answer

Sometimes we do not care how far apart two vertices are, only **whether one can be reached from the other at all**. Precisely: given a directed graph $G = (V, E)$ with $V = \{1,2,\dots,n\}$, determine whether G contains a path from i to j, for all vertex pairs $i, j \in V$.

DEFINITION. The **transitive closure** of G is the graph $G^* = (V, E^*)$ where

$$E^* = \{(i,j)\text{:there is a path from vertex } i \text{ to vertex } j \text{ in } G\}.$$

The word *transitive* comes from the corresponding property of relations: a relation R is transitive if aRb and bRc together force aRc. The transitive closure is the smallest transitive relation containing the original — in graph terms, add a shortcut edge wherever a path exists.

The first answer takes ten seconds to implement: **assign a weight of 1 to each edge of E and run the Floyd–Warshall algorithm.** If there is a path from vertex i to vertex j we get $d_{ij} < n$, because a simple path has at most $n-1$ edges and each edge now costs 1. Otherwise $d_{ij} = \infty$. The cost is $\Theta(n^3)$, and the code is the code you already have.

15.7.2 The better answer: swap the operators

A second method can save time and space in practice. It **substitutes the logical operation** $\vee$ **(logical OR) and** $\wedge$ **(logical AND) for the arithmetic operations** min **and** $+$ in the Floyd–Warshall algorithm.

This is the $(\min, +)$ semiring of §15.4.9 replaced by the boolean semiring $(\vee,\wedge)$, in which $\bar{0} =$ FALSE and $\bar{1} =$ TRUE. The substitution is forced once you see what the operators *mean*. In the distance version, min answers "which of these two routes is better?" and $+$ answers "what does it cost to do this and then that?". In the reachability version, $\vee$ answers "is either of these two routes available?" and $\wedge$ answers "can I do this **and** then that?". Same structure, different arithmetic.

Define, for $i, j, k = 1,2,\dots,n$, the value $t_{ij}^{(k)}$ to be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate vertices in the set $\{1,2,\dots,k\}$, and 0 otherwise. We construct the transitive closure $G^* = (V, E^*)$ by putting edge (i,j) into E^* if and only if $t_{ij}^{(n)} = 1$. **Figure 15.18** gives the recursive definition.

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{if } i = j \text{ or } (i,j) \in E \end{cases}$$

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 $\vee$ for min and $\wedge$ for $+$.

Written out:

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E, \\ 1 & \text{if } i = j \text{ or } (i,j) \in E, \end{cases}$$

(15.11)

and for $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}). \tag{15.12}$$

Read Equation (15.12) aloud: *j is reachable from i using only the first k vertices in the middle if either it was already reachable using only the first $k-1$, or you can get from i to k and from k to j, both using only the first $k-1$.* It is exactly the case analysis of §15.6.2 with "cheapest" replaced by "possible".

Note that the base case Equation (15.11) gives $t_{ii}^{(0)} = 1$ for every i, because a vertex reaches itself by the empty path. That is the boolean counterpart of $w_{ii} = 0$.

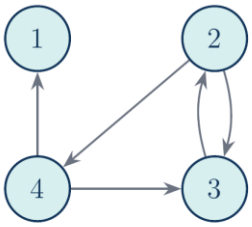
15.7.3 The algorithm and the example

Figure 15.19 gives the procedure.

```
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 15.20 shows a four-vertex graph and all five matrices.



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

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 (Figure 25.5 in the third edition of Cormen et al.).

Listing 15.8 reproduces them.

Listing 15.8 The transitive-closure algorithm in full. Compile with `g++ -std=c++17 -O2 -o transitive_closure transitive_closure.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;
}
```

Its output:

```
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)
```

All five matrices match **Figure 15.20**. Walk through the interesting rounds. Round $k = 1$ changes nothing, because vertex 1 has **no outgoing edges** — row 1 of $T^{(0)}$ is (1,0,0,0) — so $t_{1j}^{(0)} = 0$ for every $j \neq 1$ and the term $t_{i1} \wedge t_{1j}$ is always 0. Round $k = 2$ sets $t_{34}^{(2)} = 1$: vertex 3 reaches vertex 2, and vertex 2 reaches vertex 4, so $3 \to 2 \to 4$ exists. Round $k = 3$ sets $t_{42}^{(3)} = 1$ by $4 \to 3 \to 2$. Round $k = 4$ is the big one: it fills the whole of rows 2 and 3, because both now reach vertex 4, and vertex 4 reaches vertex 1. Row 1 stays (1,0,0,0) for ever, correctly: vertex 1 is a sink and reaches nothing but itself.

15.7.4 Why booleans are so much faster in practice

Both methods are $\Theta(n^3)$, so why does the boolean version save time and space? Two reasons.

Space. A distance needs a 32- or 64-bit word. A boolean needs one **bit**. Packing the matrix into machine words cuts memory by a factor of 32 or 64, which decides whether a large graph fits in cache.

Time. And once the bits are packed, the inner j-loop of line 12 becomes a bitwise OR of two whole rows — and a 64-bit processor performs 64 of those loop iterations in a single instruction. In C++ this is two lines with `std::bitset`. **Listing 15.9** measures it.

Listing 15.9 The boolean triple loop, one byte per entry and one bit per entry, measured. Compile with `g++ -std=c++17 -O2 -o ex_bitset_tc ex_bitset_tc.cpp`.

```
// Chapter 15, Exercise solution: transitive closure at 64 columns per instruction.
// The inner j-loop of TRANSITIVE-CLOSURE ORs whole rows, so a machine word can
// do 64 of its iterations at once. Same Theta(n^3) work, a much smaller constant.
#include <iostream>
#include <iomanip>
#include <vector>
#include <bitset>
#include <random>
#include <chrono>
using namespace std;
using namespace std::chrono;

const int N = 1024;                     // fixed at compile time, as std::bitset needs

int main() {
    mt19937 rng(20250915);
    uniform_int_distribution<int> pick(0, N - 1);

    vector<vector<char>>   plain(N, vector<char>(N, 0));
    vector<bitset<N>>      fast(N);

    for (int i = 0; i < N; ++i) { plain[i][i] = 1; fast[i][i] = 1; }
    for (int e = 0; e < 4 * N; ++e) {   // a sparse random digraph
        int u = pick(rng), v = pick(rng);
        plain[u][v] = 1; fast[u][v] = 1;
    }

    auto t0 = steady_clock::now();
    for (int k = 0; k < N; ++k)
        for (int i = 0; i < N; ++i)
            if (plain[i][k])
                for (int j = 0; j < N; ++j)
                    plain[i][j] = plain[i][j] | plain[k][j];
    auto t1 = steady_clock::now();

    for (int k = 0; k < N; ++k)
        for (int i = 0; i < N; ++i)
            if (fast[i][k]) fast[i] |= fast[k];        // 1024 columns in 16 words
    auto t2 = steady_clock::now();
```

Listing 15.9 (Continued) *The boolean triple loop, one byte per entry and one bit per entry, measured. Compile with g++ -std=c++17 -O2 -o ex_bitset_tc ex_bitset_tc.cpp.*

```
    bool same = true;
    long long reachable = 0;
    for (int i = 0; i < N && same; ++i)
        for (int j = 0; j < N; ++j) {
            if ((bool)plain[i][j] != fast[i][j]) { same = false; break; }
            reachable += plain[i][j];
        }

    cout << "n = " << N << ", edges = " << 4 * N << '\n';
    cout << "byte-per-entry version: " << fixed << setprecision(1)
         << duration<double, milli>(t1 - t0).count() << " ms\n";
    cout << "bitset version:         "
         << duration<double, milli>(t2 - t1).count() << " ms\n";
    cout << "the two closures agree: " << (same ? "yes" : "NO") << '\n';
    cout << "reachable pairs found:  " << reachable << " out of " << (long long)N * N <<
'\n';
    return 0;
}
```

Its output on the machine used to prepare this chapter:

```
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, with identical output. This is worth internalizing as a general lesson: asymptotic analysis tells you which algorithm to choose, and it says nothing at all about the constant factor, which here is worth nearly two orders of magnitude. Both matter; neither substitutes for the other.

15.8 Johnson's Algorithm for Sparse Graphs

15.8.1 The wish

Floyd–Warshall costs $\Theta(V^3)$ whether the graph has a million edges or nine. On a **sparse** graph — one where E is much smaller than V^2, which describes almost every graph that arises from a road network, a web crawl, a social network or a circuit — that is wasteful. The goal is an algorithm that finds shortest paths between all pairs in

$$O(V^2 \lg V + VE)$$

time, which for sparse graphs is **asymptotically faster than either repeated squaring of matrices or the Floyd–Warshall algorithm**. It returns the matrix of shortest-path weights for all pairs of vertices, or reports that the input graph contains a negative-weight cycle. It uses as subroutines both Dijkstra's algorithm and the Bellman–Ford algorithm. And it uses the technique of **reweighting**.

Where does that bound come from? It is exactly the cost of V runs of Dijkstra with a Fibonacci heap, from **Table 15.1**. So Johnson's algorithm is, at bottom, an answer to one question: *how do we make $V \times$ Dijkstra legal on a graph with negative edges?*

15.8.2 What a reweighting must accomplish

Figure 15.21 lays out the plan and the two obligations that come with it.

- 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 $\widehat{w}$ 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 $\widehat{w}$.
  2. For every edge (u, v), the new weight $\widehat{w}(u, v)$ is non-negative.

Figure 15.21 If a new weight function $\widehat{w}$ preserves shortest paths and is non-negative everywhere, then $V \times$ Dijkstra on $\widehat{w}$ answers the original question.

The new set of edge weights $\widehat{w}$ must satisfy two properties:

1. **Shortest paths are preserved.** For all pairs of vertices $u, v \in V$, a path p is a shortest path from u to v using weight function w **if and only if** p is also a shortest path from u to v using weight function $\widehat{w}$.
2. **Non-negativity.** For all edges (u, v), the new weight $\widehat{w}(u, v)$ is non-negative.

Property 2 is what makes Dijkstra legal. Property 1 is what makes the answer *mean* something: if the reweighting changed which route was best, we would be solving a different problem and reporting the answer to the wrong question. Note that property 1 says the *paths* are preserved, not the *weights* — the weights certainly change, and §15.8.5 shows how to change them back.

A REWEIGHTING THAT FAILS. The obvious idea is to find the most negative weight $-c$ and add c to every edge. Property 2 holds. Property 1 fails badly, because a path with many edges gains c once per edge while a path with few edges gains c only a few times, so long routes are penalized. Take $1 \to 2$ of weight -2, $2 \to 3$ of weight -2, and $1 \to 3$ of weight -3: the cheapest route from 1 to 3 is the two-edge one, at cost -4 against -3 for the direct edge. The most negative weight is -3, so $c = 3$; adding it gives weights 1, 1 and 0, and now the two-edge route costs 2 while the direct edge costs 0. The shortest path has changed. The correct reweighting must add an amount that depends on the **endpoints**, not on the edge count.

15.8.3 The reweighting that works

Here is the idea, and it is the heart of the algorithm. Choose any function $h: V \to \mathbb{R}$ that assigns a number to every **vertex**, and define

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v) \tag{15.13}$$

for every edge $(u, v) \in E$. Think of $h(v)$ as an *altitude* assigned to vertex v: the new cost of an edge is its old cost plus the drop in altitude across it.

LEMMA 15.2 (reweighting preserves shortest paths). Let $h: V \to \mathbb{R}$ be any function and let $\widehat{w}$ be defined by Equation (15.13). Then for any path $p = \langle v_0, v_1, \dots, v_k \rangle$,

$$\widehat{w}(p) = w(p) + h(v_0) - h(v_k).$$

Consequently p is a shortest path from v_0 to v_k under w if and only if it is one under $\widehat{w}$; and G has a negative-weight cycle under w if and only if it has one under $\widehat{w}$.

Proof. Sum Equation (15.13) along the path:

$$\begin{aligned}
\widehat{w}(p) &= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) = \sum_{i=1}^{k} \left(w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)\right) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + \sum_{i=1}^{k} \left(h(v_{i-1}) - h(v_i)\right) \\
&= w(p) + h(v_0) - h(v_k).
\end{aligned}$$

The second sum **telescopes**: writing it out gives $(h(v_0) - h(v_1)) + (h(v_1) - h(v_2)) + \cdots + (h(v_{k-1}) - h(v_k))$, and every interior term appears once positively and once negatively, so all of them cancel and only $h(v_0) - h(v_k)$ survives.

Now take any two paths p and q from u to v. Both gain the *same* constant $h(u) - h(v)$, which depends only on the endpoints and not on the route. So $\widehat{w}(p) - \widehat{w}(q) = w(p) - w(q)$: the

ordering of routes between a fixed pair is untouched. One is shorter than another under $\widehat{w}$ exactly when it was shorter under w.

For the cycle claim, take p to be a cycle, so $v_0 = v_k$. Then $h(v_0) - h(v_k) = 0$ and $\widehat{w}(p) = w(p)$ exactly: cycles keep their weight under any reweighting, so negative cycles cannot be created or destroyed. ■

This proves property 1 for *every* choice of h. The whole art is in choosing an h that also delivers property 2.

15.8.4 Choosing h: the auxiliary vertex

Property 2 demands $\widehat{w}(u,v) = w(u,v) + h(u) - h(v) \geq 0$ for every edge, which rearranges to

$$h(v) \leq h(u) + w(u,v) \quad \text{for every edge } (u,v) \in E.$$

But that is a familiar condition! It is exactly the statement that no edge can be relaxed — the edge form of the **triangle inequality**, proved in Chapter 13 as Corollary 13.3: $\delta(s,v) \leq \delta(s,u) + w(u,v)$ for every edge (u,v) and every source s. So if we set $h(v) = \delta(s,v)$ for some source s, property 2 comes for free.

There is one difficulty: no single vertex of G need reach all the others, and $\delta(s,v) = \infty$ for an unreachable v makes Equation (15.13) meaningless. The fix is line 1 of JOHNSON in §15.8.5: **invent a new vertex.**

Construct $G' $ with

$$G'.V = G.V \cup \{s\}, \qquad G'.E = G.E \cup \{(s,v) : v \in G.V\}, \qquad w(s,v) = 0 \text{ for all } v \in G.V,$$

where s is a brand-new vertex not in $G.V$. Now s reaches everything in one hop, so every $\delta(s,v)$ is finite (unless there is a negative cycle, which we detect). And s has **no incoming edges**, so no shortest path between two original vertices can pass through it: to leave s you must have entered it, and nothing enters it. Adding s therefore changes no distance among the original vertices. Set

$$h(v) = \delta(s,v) \quad \text{for all } v \in G'.V,$$

computed by one run of Bellman–Ford from s — which is also exactly the test for a negative-weight cycle.

WHY THIS h MAKES EVERY WEIGHT NON-NEGATIVE. For any edge (u,v) of G, the triangle inequality on G' gives $\delta(s,v) \leq \delta(s,u) + w(u,v)$, i.e. $h(v) \leq h(u) + w(u,v)$, i.e. $0 \leq w(u,v) + h(u) - h(v) = \widehat{w}(u,v)$. That is the entire argument, and it is one line long once you have seen that the requirement *is* the triangle inequality.

15.8.5 The algorithm

Figure 15.22 gives JOHNSON(G,w).

```
JOHNSON(G, w)
1  compute G′, where G′.V = G.V ∪ {s},
       G′.E = G.E ∪ {(s, v) : v ∈ G.V}, and w(s, v) = 0 for all v ∈ 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 ∈ G′.V
5          set h(v) to the value of δ(s, v) computed by BELLMAN-FORD
6      for each edge (u, v) ∈ G′.E
7          ŵ(u, v) = w(u, v) + h(u) − h(v)
8      let D = (d_uv) be a new n × n matrix
9      for each vertex u ∈ G.V
10         run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ G.V
11         for each vertex v ∈ G.V
12             d_uv = δ̂(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 is the step that undoes the reweighting. Dijkstra on $\widehat{w}$ returns $\hat{\delta}(u, v)$, the shortest-path weight under the new weights. By Lemma 15.2 applied to a shortest path p from u to v,

$$\hat{\delta}(u, v) = \delta(u, v) + h(u) - h(v), \qquad \text{so} \qquad \begin{aligned} d_{uv} &= \delta(u, v) \\ &= \hat{\delta}(u, v) + h(v) - h(u). \end{aligned}$$

(15.14)

Note the reversed signs: to recover the original we subtract what we added.

The running time. Line 1 builds G' in $O(V)$ time. Line 2 is one Bellman–Ford run on G', which has $V + 1$ vertices and $E + V$ edges, costing $O(VE)$. Lines 4–7 touch every edge once, $O(E)$. Lines 9–11 run Dijkstra V times; with a Fibonacci heap each run costs $O(V\lg V + E)$, so the loop costs $O(V^2\lg V + VE)$, and that term dominates. Total:

$$O(V^2\lg V + VE).$$

With a binary min-heap instead, each Dijkstra costs $O(E\lg V)$ and the total is $O(VE\lg V)$ — usually the better choice in practice, and what **Listing 15.10** uses.

15.8.6 A small reweighting worked in full

Before the five-vertex graph, a four-vertex one gives the clearest possible illustration. The graph, **Figure 15.23**, has vertices numbered 0,1,2,3 and edges 0 → 1 of weight −5, 0 → 2 of weight 2, 0 → 3 of weight 3, 1 → 2 of weight 4, and 2 → 3 of weight 1.

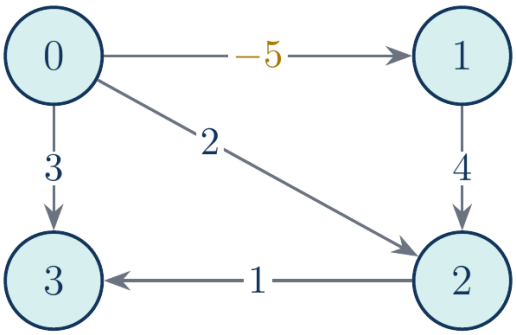


Figure 15.23 A four-vertex graph with one negative edge, $0 \to 1$ of weight -5. Dijkstra is not allowed here.

Step one: add a source s and add edges from s to all vertices of the original graph. Here s is a new vertex numbered 4. The result is **Figure 15.24**, with four new zero-weight edges drawn in a different color.

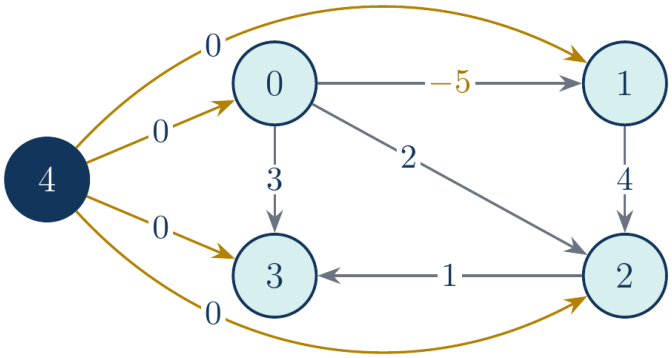


Figure 15.24 The same graph with the auxiliary vertex 4 added, joined to every original vertex by an edge of weight 0. Vertex 4 has no incoming edges, so it cannot appear inside any shortest path between original vertices.

Step two: run Bellman–Ford from vertex 4 to get h:

- Edge (4,0): $h[0] = 0$.
- Edge (4,1): $0 + (-5) = -5$, so $h[1] = -5$ — the cheapest way to reach vertex 1 is $4 \to 0 \to 1$, not the direct zero-weight edge.
- Edge (4,2): $0 + (-5) + 4 = -1$, so $h[2] = -1$, via $4 \to 0 \to 1 \to 2$.
- Edge (4,3): $h[3] = 0$.

giving $h[\,] = \{0, -5, -1, 0\}$.

Step three: remove the source and reweight with $\widehat{w}(u, v) = w(u, v) + h[u] - h[v]$. Edge by edge, the arithmetic is:

- (0,1): $-5 + 0 - (-5) = 0$.
- (0,2): $2 + 0 - (-1) = 3$.
- (0,3): $3 + 0 - 0 = 3$.
- (1,2): $4 + (-5) - (-1) = 0$.
- (2,3): $1 + (-1) - 0 = 0$.

Figure 15.25 pictures the result, with each new weight in blue above the arithmetic that produced it in red.

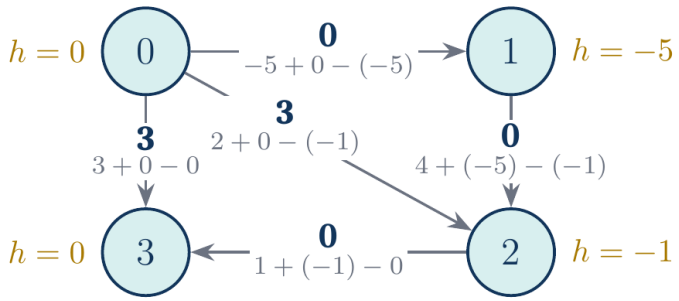


Figure 15.25 The four-vertex graph after reweighting. Every weight is now non-negative — three of the five are exactly zero — so Dijkstra's algorithm can be used from every vertex as the source.

Two details are worth noticing, because they recur in every example. First, the negative edge $0 \to 1$ became exactly 0, not merely non-negative. That is not luck: the edge lies *on* a shortest path from the source, so the triangle inequality holds with equality, so $\widehat{w}$ is exactly zero. In general, **every edge on some shortest path from s gets reweighted to zero**, and conversely — so the zero-weight edges of $\widehat{w}$ are exactly the edges usable in shortest routes out of s. (They need not form a *tree*; Exercise 15.17 shows why not.) Second, the weights did not all shrink: (0,2) went from 2 up to 3. Reweighting is not a uniform discount; it is a redistribution, and Lemma 15.2 is the guarantee that the redistribution is fair.

15.8.7 The five-vertex running example, reweighted

Now the running example. **Listing 15.10** is the full implementation.

Listing 15.10 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`.

```
// 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 15.10 (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 15.10 (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;
}
```

Its output:

```
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
```

Every number here can be checked by hand: $h[\,] = \{0, -1, -5, 0, -4\}$ and the nine reweighted values 4,13,0,0,10,0,2,0,2. All nine are non-negative, as Lemma 15.2 and §15.8.4 promised. Four of them are zero, and those four edges — $1 \to 5$, $2 \to 4$, $3 \to 2$, $4 \to 3$ — are precisely the edges usable in a shortest route out of the auxiliary source, exactly as §15.8.6 predicted. Exercise 15.17 asks you to check them one by one and to notice that, because this graph has a zero-weight cycle, they do not quite form a tree. And the resulting matrix D is the same matrix that Floyd–Warshall produced in §15.6.8 and that the slow matrix method produced in §15.4.11.

Figure 15.26 shows the whole algorithm in one picture, from G' with the original weights, through the reweighted graph, to the five Dijkstra runs.

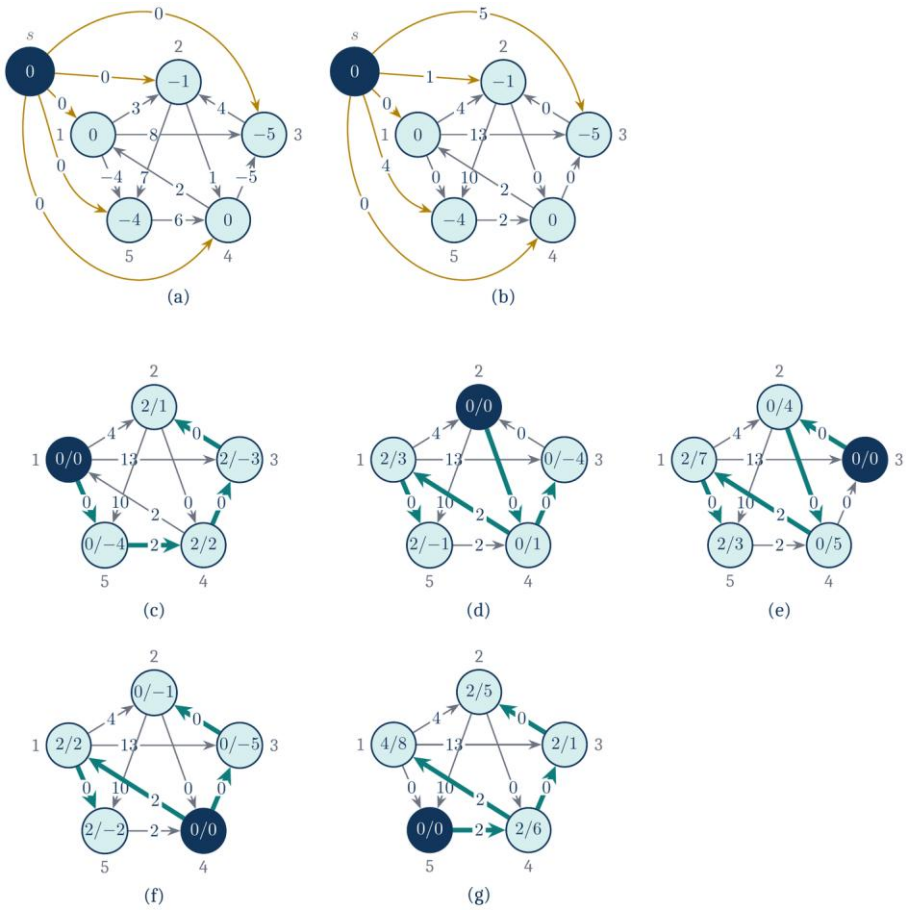


Figure 15.26 Johnson's algorithm on the graph of **Figure 15.2** (Figure 25.6 in the third edition of Cormen et al.). **(a)** G' with the original weights; the new vertex s is navy and each vertex v shows $h(v) = \delta(s, v)$. **(b)** After reweighting with $\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$; every weight is now non-negative. **(c)–(g)** Dijkstra run from each vertex in turn, the source navy and the shortest-paths tree drawn in bold teal; each vertex shows $\hat{\delta}(u, v)$ and $\delta(u, v)$ separated by a slash, related by $d_{uv} = \delta(u, v) = \hat{\delta}(u, v) + h(v) - h(u)$.

Check one entry of part (c) against Equation (15.14). The source is vertex 1, so $h(u) = h(1) = 0$. Vertex 3 shows $2/-3$: Dijkstra on $\widehat{w}$ found $\hat{\delta}(1,3) = 2$, and $h(3) = -5$, so $\delta(1,3) = 2 + (-5) - 0 = -3$. That is the entry in row 1, column 3 of the printed matrix. The reweighted graph's answer of 2 is not the answer to any question anybody asked; Equation (15.14) is what turns it back into one.

15.9 Which Algorithm, and When

15.9.1 The four costs side by side

Table 15.4 collects everything.

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

The asymptotic comparison between the two bold rows is simple once written down. Floyd–Warshall is $\Theta(V^3)$ regardless of E; Johnson is $O(V^2\lg V + VE)$. Johnson is asymptotically better exactly when $VE = o(V^3)$, that is, when

$$E = o(V^2).$$

Any graph whose edge count is genuinely sub-quadratic — a road network, where every junction has a handful of exits; a website, where every page links to a few dozen others; a social graph, where degree is bounded by human patience — favors Johnson. A graph where nearly every pair is joined favors Floyd–Warshall.

15.9.2 The crossover, measured

Asymptotics do not fix the constant, so the honest way to find the crossover is to measure it. **Listing 15.11** sweeps the density at a fixed $V = 600$.

Listing 15.11 Floyd–Warshall against $V \times$ Dijkstra at nine densities, $V = 600$ throughout. Compile with `g++ -std=c++17 -O2 -o crossover crossover.cpp`.

```
// Chapter 15, Listing: measuring where Johnson overtakes Floyd-Warshall.
// Random graphs with nonnegative weights, n fixed, density swept.
#include <iostream>
#include <iomanip>
#include <vector>
#include <queue>
#include <random>
#include <chrono>
using namespace std;
using namespace std::chrono;

const long long INF = 1000000000LL;
struct Edge { int u, v; long long w; };

double timeFloydWarshall(int n, const vector<Edge>& E) {
    vector<vector<long long>> D(n, vector<long long>(n, INF));
    for (int i = 0; i < n; ++i) D[i][i] = 0;
    for (const Edge& e : E) D[e.u][e.v] = min(D[e.u][e.v], e.w);
    auto t0 = steady_clock::now();
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i) {
            if (D[i][k] >= INF) continue;              // nothing can improve through k
            for (int j = 0; j < n; ++j)
                if (D[k][j] < INF && D[i][k] + D[k][j] < D[i][j])
                    D[i][j] = D[i][k] + D[k][j];
        }
    auto t1 = steady_clock::now();
    return duration<double, milli>(t1 - t0).count();
}

double timeJohnson(int n, const vector<Edge>& E) {
    // Weights are already nonnegative here, so h is identically 0 and the
    // Bellman-Ford pass is a formality; we time |V| runs of Dijkstra.
    vector<vector<pair<int,long long>>> adj(n);
    for (const Edge& e : E) adj[e.u].push_back({e.v, e.w});
    auto t0 = steady_clock::now();
    vector<long long> d(n);
    long long checksum = 0;
    for (int s = 0; s < n; ++s) {
        fill(d.begin(), d.end(), 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;
```

Listing 15.11 (Continued) *Floyd–Warshall against V\times Dijkstra at nine densities, V = 600 throughout. Compile with g++ -std=c++17 -O2 -o crossover crossover.cpp.*

```
            for (auto [v, w] : adj[u])
                if (d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
        }
        for (int v = 0; v < n; ++v) if (d[v] < INF) checksum += d[v];
    }
    auto t1 = steady_clock::now();
    if (checksum == -1) cout << "";                    // keep the work from being elided
    return duration<double, milli>(t1 - t0).count();
}

int main() {
    const int n = 600;
    mt19937 rng(20250915);
    uniform_int_distribution<int> weight(1, 1000);
    uniform_int_distribution<int> vertex(0, n - 1);

    cout << "n = " << n << ", weights in [1,1000], one random graph per row\n\n";
    cout << setw(10) << "|E|" << setw(12) << "|E|/V"
         << setw(16) << "Floyd (ms)" << setw(16) << "Johnson (ms)"
         << setw(10) << "winner" << '\n';

    for (long long m : {1200LL, 3000LL, 6000LL, 12000LL, 24000LL, 48000LL,
                        96000LL, 180000LL, 360000LL}) {
        vector<Edge> E;
        E.reserve(m);
        for (long long t = 0; t < m; ++t) {
            int u = vertex(rng), v = vertex(rng);
            if (u == v) { --t; continue; }
            E.push_back({u, v, weight(rng)});
        }
        double fw = timeFloydWarshall(n, E);
        double jo = timeJohnson(n, E);
        cout << setw(10) << m << setw(12) << fixed << setprecision(1) << (double)m / n
             << setw(16) << setprecision(1) << fw
             << setw(16) << setprecision(1) << jo
             << setw(10) << (jo < fw ? "Johnson" : "Floyd") << '\n';
    }
    return 0;
}
```

Its output on the machine used to prepare this chapter:

```
n = 600, weights in [1,1000], one random graph per row

       |E|       |E|/V      Floyd (ms)    Johnson (ms)    winner
      1200         2.0            21.6            12.9   Johnson
      3000         5.0            86.1            38.2   Johnson
      6000        10.0            99.3            56.7   Johnson
     12000        20.0           103.5            78.3   Johnson
     24000        40.0           112.5           103.1   Johnson
     48000        80.0           113.6           133.2     Floyd
     96000       160.0           115.7           171.7     Floyd
    180000       300.0           116.4           217.6     Floyd
    360000       600.0           118.6           301.5     Floyd
```

The crossover sits at about $|E|/|V| \approx 80$, that is $|E| \approx 48{,}000$ against a maximum possible $600 \times 599 = 359{,}400$ — a density of roughly **13 percent**. Below that, run Johnson; above it, run Floyd–Warshall.

Two features of the table deserve comment. From $|E|/|V| = 10$ upward, Floyd–Warshall's time is essentially flat — 99 ms to 119 ms across a sixty-fold change in edge count — exactly as $\Theta(V^3)$ predicts. The two sparsest rows are faster only because of the `continue` in `timeFloydWarshall`,

which skips a whole inner loop whenever $d_{ik} = \infty$, and on a barely-connected graph that skip is available most of the time. Johnson's time, by contrast, grows steadily with E, as $O(VE\lg V)$ predicts. And at the sparsest row Johnson is about 1.7 times faster — precisely the regime where a real road network lives.

15.9.3 A decision procedure

Put together, **Table 15.5** is what to do when someone hands you a graph.

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 $n \leq 400$), 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?	$V \times$ Dijkstra with a binary heap	Johnson: one Bellman–Ford, reweight, then $V \times$ 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	—

15.10 Profitina in Practice

15.10.1 Where all-pairs distances appear in a GEO platform

Profitina is an AI-visibility platform: it scans a website, scores how ready that site is to be **cited by an AI answer engine**, and explains each signal it measured [9]. That work is a graph problem more often than it looks.

A crawled site *is* a directed graph. Pages are vertices; links are edges. The number of clicks from the home page to a given page — its **click depth** — is a shortest-path distance with every edge of weight 1, and it matters because a page buried six clicks deep is a page an answer engine is unlikely ever to reach, however good its content. Weighted variants are more informative still: give an edge a small weight when the link is prominent and a larger one when it is buried in a footer, and the shortest-path distance becomes a measure of how hard a page is to *find*, not merely how far away it is.

Two questions the platform's analytics must answer are genuinely all-pairs questions rather than single-source ones.

The diameter of the internal link graph. The largest $\delta(u, v)$ over all pairs is the worst-case journey inside the site. A large diameter is a structural warning: some part of the site is reachable only through a long chain, and reorganizing the navigation would help more than rewriting the copy. Finding the largest element of a set requires computing the set, which is the all-pairs problem. For a site section of a few hundred pages, Floyd–Warshall answers it in milliseconds.

Reachability, and the orphan report. Which pages can be reached from the home page at all? Which pages can reach the sitemap? That is the transitive closure of §15.7, and the bit-packed version of **Listing 15.9** handles tens of thousands of pages comfortably. A page that appears in no row of the closure except its own is an **orphan** — present on the server, invisible to a crawler, and therefore invisible to an answer engine. This is one of the cheapest and most valuable checks a scanner can run.

Two cautions, both reflecting Profitina's own rules. First, the platform **advises; the user decides** — a large diameter is a finding to explain, never a guarantee of anything. Second, a real site quickly grows past the point where an $n \times n$ matrix is affordable: at 100,000 pages the output alone is eighty gigabytes (§15.2.4). Production systems therefore run all-pairs algorithms on *sections* — a subgraph of a few hundred or a few thousand pages — and use single-source algorithms from a handful of designated roots for the whole site. Knowing which regime you are in is the practical half of this chapter.

15.10.2 The worked case study: Profitina Laundry

The runnable case study in this book is always **Profitina Laundry**, a teaching example and not the real product.

Profitina Laundry collects and delivers across eight parts of Surabaya: Keputih, Gebang, Klampis, Manyar, Ngagel, Darmo, Wonokromo and Rungkut. The road segments between them are **one-way in places**, so the driving-time matrix is not symmetric — the run from Keputih to Manyar takes nine minutes while the return takes twelve. The owner has two questions, and they are not the same question.

1. *Where should the depot be, if the promise to customers is "we will reach you within X minutes"?* That asks for the site whose **worst** outbound run is smallest — the **center** of the graph, the vertex of least **eccentricity**, where the eccentricity of v is $\max_j \delta(v, j)$.
2. *Where should the depot be, if the aim is to minimize total driving?* That asks for the site with the smallest **sum** of out-and-back times.

Both need the full all-pairs matrix. **Listing 15.12** computes it with Floyd–Warshall, carrying Π so that the routes can be read back.

Listing 15.12 Profitina Laundry: all-pairs driving times, eccentricity, the graph center, the diameter, and the route behind the worst run. Compile with `g++ -std=c++17 -O2 -o laundry laundry.cpp`.

```
// Chapter 15, Listing: Profitina Laundry -- choosing the depot from an all-pairs
// distance matrix. Floyd-Warshall on a one-way road network, then eccentricity,
// the graph center, and the diameter.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;

const long long INF = 1000000000LL;
const int NIL = -1;

int main() {
    // Eight places in the delivery network. Times are driving minutes on one-way
    // segments, so the matrix is NOT symmetric: 0 -> 1 may cost more than 1 -> 0.
    vector<string> name = {
        "Keputih", "Gebang", "Klampis", "Manyar", "Ngagel", "Darmo", "Wonokromo", "Rungkut"
    };
    const int n = 8;

    // Directed segments (from, to, minutes).
    vector<vector<long long>> road = {
        {0, 1,  4}, {1, 0,  5},
        {0, 2,  9}, {2, 0,  8},
        {1, 2,  3}, {2, 1,  3},
        {1, 3,  7},
        {2, 3,  2}, {3, 2,  4},
        {3, 4,  6}, {4, 3,  6},
        {2, 5, 14},
        {4, 5,  3}, {5, 4,  4},
        {5, 6,  5}, {6, 5,  5},
        {4, 6,  9},
        {6, 0, 20},
        {6, 7,  6}, {7, 6,  6},
        {0, 7,  9}, {7, 0,  9}
    };

    vector<vector<long long>> d(n, vector<long long>(n, INF));
    vector<vector<int>>       pi(n, vector<int>(n, NIL));
    for (int i = 0; i < n; ++i) d[i][i] = 0;
    for (auto& r : road) {
        int u = (int)r[0], v = (int)r[1];
        if (r[2] < d[u][v]) { d[u][v] = r[2]; pi[u][v] = u; }
    }
```

Listing 15.12 (Continued) *Profitina Laundry: all-pairs driving times, eccentricity, the graph center, the diameter, and the route behind the worst run. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
// FLOYD-WARSHALL, in place, carrying the predecessor matrix along.
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];
                pi[i][j] = pi[k][j];
            }

cout << "All-pairs driving times (minutes)\n\n" << setw(11) << " ";
for (int j = 0; j < n; ++j) cout << setw(11) << name[j];
cout << '\n';
for (int i = 0; i < n; ++i) {
    cout << setw(11) << name[i];
    for (int j = 0; j < n; ++j) {
        if (d[i][j] >= INF) cout << setw(11) << "-";
        else                cout << setw(11) << d[i][j];
    }
    cout << '\n';
}

// The eccentricity of v is the largest shortest-path distance from v to anywhere.
// The center is the vertex of least eccentricity: the best depot under a
// worst-case service promise. The diameter is the largest eccentricity.
cout << "\nEccentricity of each candidate depot (worst outbound run):\n";
long long best = INF, diameter = 0;
int center = NIL;
for (int v = 0; v < n; ++v) {
    long long ecc = 0;
    for (int j = 0; j < n; ++j) ecc = max(ecc, d[v][j]);
    cout << "   " << setw(11) << name[v] << setw(5) << ecc << " min\n";
    if (ecc < best) { best = ecc; center = v; }
    diameter = max(diameter, ecc);
}
cout << "\ncenter  = " << name[center] << " (eccentricity " << best << " min)\n";
cout << "diameter = " << diameter << " min\n";
```

Listing 15.12 (Continued) *Profitina Laundry: all-pairs driving times, eccentricity, the graph center, the diameter, and the route behind the worst run. Compile with g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
    // Total round-trip cost of serving every point from a fixed depot: sum of
    // out-and-back times. A different question, and often a different answer.
    cout << "\nSum of out-and-back times from each candidate depot:\n";
    long long bestSum = INF; int bestHub = NIL;
    for (int v = 0; v < n; ++v) {
        long long total = 0;
        for (int j = 0; j < n; ++j) total += d[v][j] + d[j][v];
        cout << "   " << setw(11) << name[v] << setw(5) << total << " min\n";
        if (total < bestSum) { bestSum = total; bestHub = v; }
    }
    cout << "\nlowest total round-trip: " << name[bestHub] << " (" << bestSum << " min)\n";

    // The predecessor matrix still holds the routes, not just the times.
    cout << "\nThe worst run out of the center, read back from Pi:\n   ";
    int worst = 0;
    for (int j = 0; j < n; ++j) if (d[center][j] > d[center][worst]) worst = j;
    vector<string> route;
    for (int v = worst; v != NIL; v = pi[center][v]) {
        route.push_back(name[v]);
        if (v == center) break;
    }
    for (int t = (int)route.size() - 1; t >= 0; --t)
        cout << route[t] << (t ? " -> " : "");
    cout << "  (" << d[center][worst] << " min)\n";
    return 0;
}
```

Its output:

```
All-pairs driving times (minutes)

               Keputih     Gebang    Klampis     Manyar     Ngagel      Darmo  Wonokromo
Rungkut
    Keputih          0          4          7          9         15         18         15
9
     Gebang          5          0          3          5         11         14         19
14
    Klampis          8          3          0          2          8         11         16
17
     Manyar         12          7          4          0          6          9         14
20
     Ngagel         18         13         10          6          0          3          8
14
      Darmo         20         17         14         10          4          0          5
11
  Wonokromo         15         19         19         15          9          5          0
6
    Rungkut          9         13         16         18         15         11          6
0

Eccentricity of each candidate depot (worst outbound run):
       Keputih   18 min
        Gebang   19 min
       Klampis   17 min
        Manyar   20 min
        Ngagel   18 min
         Darmo   20 min
     Wonokromo   19 min
       Rungkut   18 min

center  = Klampis (eccentricity 17 min)
diameter = 20 min

Sum of out-and-back times from each candidate depot:
       Keputih  164 min
        Gebang  147 min
       Klampis  138 min
        Manyar  137 min
        Ngagel  140 min
         Darmo  152 min
     Wonokromo  171 min
       Rungkut  179 min

lowest total round-trip: Manyar (137 min)

The worst run out of the center, read back from Pi:
   Klampis -> Keputih -> Rungkut  (17 min)
```

The two questions have two different answers. Put the depot in **Klampis** and no customer is more than seventeen minutes away — the strongest promise available. Put it in **Manyar** and total daily driving falls to 137 minutes, one fewer than Klampis, but the worst run stretches to twenty minutes. Which is right depends on whether the business is selling a guarantee or minimizing fuel, and the algorithm cannot make that choice. What the algorithm can do — and what it has done here — is replace an argument with a table.

Notice also the asymmetry the one-way segments produce. Keputih reaches Manyar in nine minutes but Manyar needs twelve to get back; Klampis reaches Wonokromo in sixteen but the return takes nineteen; Manyar reaches Rungkut in twenty while Rungkut reaches Manyar in eighteen. Some pairs happen to agree — Keputih and Wonokromo are fifteen minutes apart either way — but even there the routes differ: the one-way segment from Wonokromo to Keputih takes twenty minutes, and the fifteen-minute run goes round through Rungkut instead. An

undirected model would have silently averaged all of this away and produced a plausible, wrong answer. The directed all-pairs matrix keeps them apart.

Finally, the last block shows Π earning its keep: the worst run out of Klampis is not a single road but the two-leg route Klampis → Keputih → Rungkut. Without the predecessor matrix we would know the seventeen minutes and not the road.

15.11 Common Mistakes, and How to Avoid Them

Writing the loops in the wrong order. `for (i) for (j) for (k)` instead of `for (k) for (i) for (j)`. It compiles, it runs, and it is wrong; **Listing 15.4** shows it reporting ∞ where the answer is 3. Remember: k **goes first**, because the recurrence reads round-$(k-1)$ values and only the outermost position guarantees that.

Flattening the superscripts. Writing $l_{ij}^{(m)}$ as `l(m)ij` or `lij^m`, or $d_{ij}^{(k)}$ as `dij^k`. The parenthesized superscript is a *stage label*, not an exponent, and running it together with the subscripts destroys the one thing the notation exists to make visible: which matrix in the sequence you are talking about. In this chapter, $l_{ij}^{(m)}$ has a budget of m **edges** while $d_{ij}^{(k)}$ has a permitted intermediate set of k **vertices**; conflating the two makes §15.4 and §15.6 look like the same algorithm, which they are not.

Computing `INF + INF`. If ∞ is represented as `INT_MAX`, then `INF + INF` overflows to a negative number and the algorithm cheerfully reports a route of weight −2 through two edges that do not exist. Either guard the addition — as every listing in this chapter does — or choose a sentinel small enough that twice it still fits, which is why `1000000000LL` is used here rather than `LLONG_MAX`.

Setting $\pi_{ij} = k$ instead of $\pi_{ij} = \pi_{kj}$. Vertex k is in the middle of the improved route, not at its end. The predecessor of j is the vertex before j, which on the tail segment $k \rightsquigarrow j$ is π_{kj}. Getting this wrong produces a Π whose printed "routes" are not paths in the graph. Test path reconstruction by summing the edge weights along the printed route and comparing with d_{ij} — §15.6.11 does exactly that.

Using Dijkstra because the test graph happened to work. **Listing 15.1** makes the point: on the running example, textbook Dijkstra produced twenty-five correct answers out of twenty-five, and on a four-vertex graph it produced two wrong ones out of sixteen. A negative edge invalidates Dijkstra's correctness proof whether or not it happens to break your example. Reweight first (§15.8), or use Floyd–Warshall.

Forgetting that reweighting must be undone. Dijkstra on $\widehat{w}$ returns $\hat{\delta}(u,v)$, which is *not* the answer. Equation (15.14), $d_{uv} = \hat{\delta}(u,v) + h(v) - h(u)$, converts it back. Omitting line 12 of **Figure 15.22** gives a matrix of entirely non-negative numbers, which on a graph with negative edges should immediately look suspicious.

Adding a constant to every edge to remove negative weights. The trap called out in §15.8.2: it penalizes paths in proportion to their edge count and changes which path is shortest. The correct reweighting depends on the endpoints, through a vertex function h, which is exactly why Lemma 15.2 telescopes and the naive fix does not.

Assuming D is symmetric. In a directed graph $\delta(u, v)$ and $\delta(v, u)$ are unrelated; the laundry network of §15.10.2 has pairs differing by three minutes. Only in an undirected graph, where every edge is usable both ways at the same cost, is D symmetric.

Running an all-pairs algorithm when you needed one source. If the question is "how far is everything from here?", one run of Dijkstra or Bellman–Ford answers it in a fraction of the time and a fraction of the memory. All-pairs is for when you genuinely need the whole table.

Trusting the off-diagonal entries when a negative cycle exists. Once $d_{ii} < 0$ for some i, the true answers involving that component are $-\infty$ and the finite numbers in the matrix are artifacts of how many rounds happened to run. Check the diagonal (§15.6.14) *before* reading anything else.

15.12 Chapter Summary

The **all-pairs shortest-paths problem** asks for $\delta(i, j)$ for every ordered pair of vertices, together with a route achieving it. The input is the adjacency matrix $W = (w_{ij})$ of Equation (15.1), with 0 on the diagonal and ∞ for a missing edge. The output is a distance matrix and a **predecessor matrix** $\Pi = (\pi_{ij})$, from which PRINT-ALL-PAIRS-SHORTEST-PATH recovers any route in time linear in its length.

The **naive** solution runs a single-source algorithm n times: $O(V^3)$ with Dijkstra when every weight is non-negative, $O(V^2E)$ — that is $O(V^4)$ on a dense graph — with Bellman–Ford when they are not.

The **matrix-multiplication** method characterizes a shortest path by the number of edges it uses. With $l_{ij}^{(m)}$ the least weight of a path from i to j using at most m edges, and $l_{ij}^{(0)} = 0$ for $i = j$ and ∞ otherwise, the recurrence is $l_{ij}^{(m)} = \min(l_{ij}^{(m-1)}, \min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\})$, which collapses to $\min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}$ because $w_{jj} = 0$. Since a shortest path is simple and so has at most $n - 1$ edges, $\delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots$. One step of the recurrence is EXTEND-SHORTEST-PATHS, which is the schoolbook matrix product with min for $+$ and $+$ for $\times$ — the $(\min, +)$ semiring — so $L^{(m)} = W^m$ and SLOW-APSP costs $\Theta(n^4)$. Because $\odot$ is associative and the sequence is constant past $n - 1$, **repeated squaring** reaches $L^{(2^{\lceil \lg(n-1)\rceil})} = W^{2^{\lceil \lg(n-1)\rceil}}$ in only $\lceil \lg(n - 1)\rceil$ products, for $\Theta(n^3 \lg n)$.

The **Floyd–Warshall** algorithm characterizes a shortest path instead by which vertices it may pass through. With $d_{ij}^{(k)}$ the least weight of a path from i to j whose intermediate vertices lie in $\{1, \ldots, k\}$, the recurrence is $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ with $d_{ij}^{(0)} = w_{ij}$, and $d_{ij}^{(n)} = \delta(i, j)$. Because the two candidates are only two, the triple loop is $\Theta(n^3)$ — and k **must** be the outermost loop, because every value read must belong to the previous round. The optimization to a single in-place matrix is safe, because round k never changes row k or column k. A negative entry on the diagonal afterward is exactly the condition that a negative-weight cycle exists.

Transitive closure is the same triple loop over the boolean semiring: $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$, with $t_{ij}^{(0)} = 1$ when $i = j$ or $(i, j) \in E$. It is $\Theta(n^3)$, and packing the rows into machine words gives a measured speedup of more than seventy.

Johnson's algorithm makes $V \times$ Dijkstra legal on a graph with negative edges. Adding an auxiliary source s joined to every vertex by a zero-weight edge, one Bellman–Ford run gives

$h(v) = \delta(s, v)$; reweighting by $\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$ leaves every shortest path a shortest path, because $\widehat{w}(p) = w(p) + h(v_0) - h(v_k)$ telescopes, and makes every weight non-negative, because the requirement is exactly the triangle inequality. Then V runs of Dijkstra and the conversion $d_{uv} = \hat{\delta}(u, v) + h(v) - h(u)$ give the answer in $O(V^2 \lg V + VE)$.

Which to use. Floyd–Warshall when the graph is dense or small or the code must be readable; Johnson when it is sparse. The measured crossover, at $V = 600$, is about thirteen percent density.

15.12.1 Looking ahead

Chapter 16, Evaluation and Discussion IV, works problems on Chapters 13 to 15 and on the book as a whole. Every algorithm in this chapter appears there, and two kinds of problem deserve rehearsal because this chapter answered them with programs rather than assertions: *trace Floyd–Warshall on this graph, showing every* $D^{(k)}$, and *why must k be the outermost loop?*

15.13 Exercises

15.1 Write down the matrix W of Equation (15.1) for the directed graph on vertices $\{1,2,3\}$ with edges $1 \to 2$ of weight 4, $2 \to 3$ of weight -2 and $3 \to 1$ of weight 5. Then compute $L^{(2)}$ and $L^{(3)}$ by hand and state $\delta(i, j)$ for all nine pairs.

15.2 Compute row 3 of $L^{(2)}$ for the five-vertex running example by hand, showing all five candidates for each of the five entries, and check your answer against §15.4.11.

15.3 Explain, in two sentences, what would go wrong with the simplification from Equation (15.4) to Equation (15.5) if the definition of W had put ∞ rather than 0 on the diagonal. Would SLOW-APSP still be correct if we kept the unsimplified recurrence of Equation (15.4) and ∞ on the diagonal?

15.4 Prove that the $\odot$ product of EXTEND-SHORTEST-PATHS is associative, that is, $(A \odot B) \odot C = A \odot (B \odot C)$ for all $n \times n$ matrices over $\mathbb{R} \cup \{\infty\}$. Which section of this chapter depends on the result?

15.5 Show that the matrix $L^{(0)}$ of Equation (15.3) — zeros on the diagonal, ∞ elsewhere — is a two-sided identity for $\odot$. Why is this the right analog of the ordinary identity matrix?

15.6 How many calls to EXTEND-SHORTEST-PATHS does SLOW-ALL-PAIRS-SHORTEST-PATHS make, and how many does FASTER-ALL-PAIRS-SHORTEST-PATHS make, for $n = 5$, $n = 100$, $n = 1000$ and $n = 1026$? Give the general formulas and comment on the last case.

15.7 For the five-vertex running example, identify the one entry of the answer matrix that does *not* attain its final value until $m = 4$, name the route responsible, and explain why this single entry forbids stopping SLOW-APSP early.

15.8 Perform the round $k = 2$ of Floyd–Warshall on the running example by hand, starting from the $D^{(1)}$ printed in §15.6.8. List every cell that changes and the route responsible, and state the corresponding change to Π.

15.9 Trace graph (b) of **Listing 15.4** by hand under the loop order `for (i) for (j) for (k)`, and explain precisely why it reports a *finite but too large* value for $\delta(1,2)$ rather than the ∞ that graph (a) produces. Why is the finite wrong answer the more dangerous of the two?

15.10 Prove that in round k of Floyd–Warshall, no entry of row k or of column k changes. Where in the chapter is this used, and what hypothesis about the graph does the proof require?

15.11 Explain why the predecessor update in Equation (15.10) is $\pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$ and not $\pi_{ij}^{(k)} = k$. Construct the smallest graph on which the wrong version produces a "route" that is not a path in the graph.

15.12 Compute the **eccentricity** of every vertex of the five-vertex running example from the matrix $D^{(5)}$ of §15.6.8, and hence the **center** and the **diameter**. Is the center unique?

15.13 Run TRANSITIVE-CLOSURE on the five-vertex running example and give $T^{(0)}$ through $T^{(5)}$. At which round does the matrix first become all ones, and which vertex is responsible?

15.14 The Floyd–Warshall triple loop works over any semiring. Write one program that runs it over three: $(\min, +)$ for shortest-path weights, (max,min) for the **widest path** (the route whose narrowest edge is as wide as possible), and a $(+,\times)$ companion matrix that counts how many distinct shortest paths join each pair. Demonstrate each on a small graph and check the counts by hand.

15.15 Prove that Johnson's reweighting cannot create or destroy a negative-weight cycle, and explain why that fact is needed for line 2 of **Figure 15.22** to be a valid test.

15.16 Why can the auxiliary vertex s of Johnson's algorithm never appear as an intermediate vertex on a shortest path between two original vertices? Give the one-line reason, and say what would break if we instead chose $h(v) = \delta(u_0, v)$ for some existing vertex u_0.

15.17 On the reweighted graph of §15.8.7, exactly four edges have weight 0. Identify them, prove the general statement that $\widehat{w}(u, v) = 0$ if and only if $\delta(s, v) = \delta(s, u) + w(u, v)$, and then decide whether the zero-weight edges of this particular graph form a tree. Justify your answer.

15.18 Using **Table 15.4**, derive algebraically the density at which Johnson's $O(V^2 \lg V + VE)$ equals Floyd–Warshall's $\Theta(V^3)$, ignoring constants, and compare your prediction with the measured crossover of §15.9.2. Why do the two differ?

15.14 Solutions to the Exercises

15.1 By Equation (15.1), with ∞ where there is no edge:

$$W = \begin{pmatrix} 0 & 4 & \infty \\ \infty & 0 & -2 \\ 5 & \infty & 0 \end{pmatrix}.$$

$L^{(1)} = W$. For $L^{(2)}$, apply $l_{ij}^{(2)} = \min_k\{w_{ik} + w_{kj}\}$. Row 1: $l_{11}^{(2)} = \min(0+0,\ 4+\infty,\ \infty+5) = 0$; $l_{12}^{(2)} = \min(0+4,\ 4+0,\ \infty+\infty) = 4$; $l_{13}^{(2)} = \min(0+\infty,\ 4+(-2),\ \infty+0) = 2$. Row 2: $l_{21}^{(2)} = \min(\infty+0,\ 0+\infty,\ -2+5) = 3$; $l_{22}^{(2)} = \min(\infty+4,\ 0+0,\ -2+\infty) = 0$; $l_{23}^{(2)} = -2$. Row 3: $l_{31}^{(2)} = 5$; $l_{32}^{(2)} = \min(5+4,\ \infty+0,\ 0+\infty) = 9$; $l_{33}^{(2)} = \min(5+\infty,\ \infty+(-2),\ 0+0) = 0$. So

$$L^{(2)} = \begin{pmatrix} 0 & 4 & 2 \\ 3 & 0 & -2 \\ 5 & 9 & 0 \end{pmatrix}.$$

Since $n = 3$, $L^{(n-1)} = L^{(2)}$ is already the answer, and indeed $L^{(3)} = L^{(2)} \odot W = L^{(2)}$ (the reader should check one entry, say $l_{13}^{(3)} = \min(0+\infty,\ 4+(-2),\ 2+0) = 2$). So δ is the matrix above. Sanity: the only cycle is $1 \to 2 \to 3 \to 1$ of weight $4 - 2 + 5 = 7 > 0$, so there is no negative cycle and the diagonal is all zeros, as it must be.

15.2 Row 3 of W is $(\infty, 4, 0, \infty, \infty)$, so only $k = 2$ and $k = 3$ give finite left factors. $l_{31}^{(2)} = \min\{\infty + 0,\ 4 + \infty,\ 0 + \infty,\ \infty + 2,\ \infty + \infty\} = \infty$. $l_{32}^{(2)} = \min\{\infty + 3,\ 4 + 0,\ 0 + 4,\ \infty + \infty,\ \infty + \infty\} = 4$ — a tie between "reach 2 in one step and stay" and "stay at 3 then take the edge". $l_{33}^{(2)} = \min\{\infty + 8,\ 4 + \infty,\ 0 + 0,\ \infty + \infty,\ \infty + \infty\} = 0$. $l_{34}^{(2)} = \min\{\infty + \infty,\ 4 + 1,\ 0 + \infty,\ \infty + 0,\ \infty + 6\} = 5$, via $3 \to 2 \to 4$. $l_{35}^{(2)} = \min\{\infty + (-4),\ 4 + 7,\ 0 + \infty,\ \infty + \infty,\ \infty + 0\} = 11$, via $3 \to 2 \to 5$. Row 3 of $L^{(2)}$ is $(\infty, 4, 0, 5, 11)$, matching §15.4.11.

15.3 With ∞ on the diagonal the candidate $k = j$ would contribute $l_{ij}^{(m-1)} + \infty = \infty$ instead of $l_{ij}^{(m-1)}$, so the first argument of the outer minimum would no longer be among the inner candidates and the collapse to Equation (15.5) would be invalid. Yes, SLOW-APSP would still be correct with the *unsimplified* recurrence of Equation (15.4) — the two-argument minimum still enumerates both cases — but EXTEND-SHORTEST-PATHS would need an extra term, the resemblance to matrix multiplication would be lost, and with it the repeated-squaring trick, because $L^{(0)}$ would no longer be the identity for $\odot$.

15.4 Take any i, j. Writing out both sides,

$$\begin{aligned}((A \odot B) \odot C)_{ij} &= \min_q \left\{\min_k \{a_{ik} + b_{kq}\} + c_{qj}\right\} \\ &= \min_q \min_k \{a_{ik} + b_{kq} + c_{qj}\},\end{aligned}$$

using distributivity ($x + \min\{y, z\} = \min\{x + y, x + z\}$) to push c_{qj} inside. Similarly $(A \odot (B \odot C))_{ij} = \min_k \min_q \{a_{ik} + b_{kq} + c_{qj}\}$. Both are the minimum of the same finite set of n^2 numbers $\{a_{ik} + b_{kq} + c_{qj}\}$, taken in a different order, and the minimum of a finite set does not depend on the order in which it is scanned. Hence they are equal. §15.5 depends on this: repeated squaring is legitimate only because the bracketing may be rearranged.

15.5 $(L^{(0)} \odot A)_{ij} = \min_k \left\{l_{ik}^{(0)} + a_{kj}\right\}$. Every term with $k \neq i$ is $\infty + a_{kj} = \infty$; the term with $k = i$ is $0 + a_{ij} = a_{ij}$. So the minimum is a_{ij}. The argument on the other side is symmetric. It is the right analog because in any semiring the identity matrix carries the multiplicative identity $\bar{1}$ on the diagonal and the additive identity $\bar{0}$ elsewhere; here $\bar{1} = 0$ (since $x + 0 = x$) and $\bar{0} = \infty$ (since $\min(x, \infty) = x$), which is exactly $L^{(0)}$.

15.6 SLOW-APSP makes $n - 2$ calls (the loop runs $m = 2, \ldots, n - 1$); FASTER-APSP makes $\lceil \lg(n - 1) \rceil$. So: $n = 5$, 3 against 2; $n = 100$, 98 against 7; $n = 1000$, 998 against 10; $n = 1026$, 1024 against 11. The last case is instructive: $n = 1025$ also needs 10 (because $2^{10} = 1024 \geq 1024$) but $n = 1026$ needs 11, so a single extra vertex can add a whole matrix product — the cost is a step function in n, not a smooth one.

15.7 The entry is l_{12}, which is 3 in $L^{(1)}$, $L^{(2)}$ and $L^{(3)}$ and drops to 1 only in $L^{(4)}$. The responsible route is $1 \to 5 \to 4 \to 3 \to 2$, of weight $-4 + 6 - 5 + 4 = 1$, which uses four edges and therefore cannot be discovered with a budget of three. It forbids stopping early on a hunch. Watched on its own, the entry l_{12} sat at 3 for three consecutive matrices and looked settled — and then changed. A single entry that has stopped moving proves nothing, and neither does a small, fixed edge budget chosen because "routes are usually short". (One test *is* safe: if an entire matrix satisfies $L^{(m)} = L^{(m-1)}$, then $L^{(m+1)} = L^{(m)} \odot W = L^{(m-1)} \odot W = L^{(m)}$, so nothing will ever change again.

Here that test does not fire until $m = 5$, because $L^{(4)}$ still differs from $L^{(3)}$.) Without such a check, the safe stop is $m = n - 1$.

15.8 Starting from $D^{(1)}$ with rows $(0,3,8,\infty,-4)$, $(\infty,0,\infty,1,7)$, $(\infty,4,0,\infty,\infty)$, $(2,5,-5,0,-2)$, $(\infty,\infty,\infty,6,0)$, round $k = 2$ tests $d_{i2} + d_{2j}$ against d_{ij}.

- (1,4): $\min(\infty,\ d_{12} + d_{24} = 3 + 1 = 4) = 4$. **Changed**; route $1 \to 2 \to 4$; $\pi_{14}^{(2)} = \pi_{24}^{(1)} = 2$.
- (3,4): $\min(\infty,\ 4 + 1 = 5) = 5$. **Changed**; route $3 \to 2 \to 4$; $\pi_{34}^{(2)} = 2$.
- (3,5): $\min(\infty,\ 4 + 7 = 11) = 11$. **Changed**; route $3 \to 2 \to 5$; $\pi_{35}^{(2)} = \pi_{25}^{(1)} = 2$.

Nothing else moves: row 2 is row k; row 5 has $d_{52} = \infty$; row 4's candidates $5 + d_{2j}$ all lose ($5 + 1 = 6 > 0$ at $j = 4$, $5 + 7 = 12 > -2$ at $j = 5$). Three cells, exactly the three differences between the printed $D^{(1)}$ and $D^{(2)}$.

15.9 The true $\delta(1,2)$ is $6 + 3 + 1 = 10$, along $1 \to 4 \to 3 \to 2$. With i outermost, row 1 is filled left to right. At cell (1,2) the k-sweep can only use $k = 4$, because d_{13} is still ∞ at that moment; it finds $d_{14} + d_{42} = 6 + 6 = 12$ and writes 12. One column later, at cell (1,3), the same sweep sets $d_{13} = d_{14} + d_{43} = 6 + 3 = 9$ — but cell (1,2) has been passed and is never revisited, so the improvement $d_{13} + d_{32} = 9 + 1 = 10$ is never considered. The reported answer is 12, as the program confirms. It is the more dangerous failure because an ∞ where a route exists is obvious in any test, whereas 12 against a true 10 looks entirely reasonable: it is a real route, of a plausible length, and only a second implementation or a hand check exposes it.

15.10 During round k the candidate for d_{kj} is $d_{kk} + d_{kj}$ and the candidate for d_{ik} is $d_{ik} + d_{kk}$. Because the graph has no negative-weight cycle, $d_{kk} \geq 0$: the cheapest closed walk through k is the empty one, of weight 0. So both candidates are $\geq$ the incumbent, and neither entry changes. It is used in §15.6.13 to prove that the in-place optimization is safe, and the hypothesis required is exactly **no negative-weight cycle**.

15.11 If the minimum came from routing through k, the improved path ends with the segment $k \rightsquigarrow j$, and the vertex immediately before j is the predecessor of j *on the best route out of* k, namely $\pi_{kj}^{(k-1)}$. Vertex k sits in the middle of the new route, not at its end. The two rules differ exactly when the segment $k \rightsquigarrow j$ has more than one edge, that is, when $\pi_{kj} \neq k$. The smallest such graph has four vertices and three edges, $1 \to 3$, $3 \to 2$ and $2 \to 4$, each of weight 1. Round $k = 2$ sets $d_{34} = d_{32} + d_{24} = 2$ with $\pi_{34} = \pi_{24} = 2$. Round $k = 3$ then improves d_{14} from ∞ to $d_{13} + d_{34} = 1 + 2 = 3$. The correct rule records $\pi_{14} = \pi_{34} = 2$, and PRINT-ALL-PAIRS-SHORTEST-PATH produces $1 \to 3 \to 2 \to 4$, whose edge weights sum to $3 = d_{14}$. The wrong rule records $\pi_{14} = k = 3$, and the same procedure produces $1 \to 3 \to 4$ — not a path in the graph at all, since the edge $3 \to 4$ does not exist. Note that the distance matrix is identical either way: only the routes are ruined, which is why this bug survives any test that checks numbers and not paths.

15.12 From $D^{(5)}$, the eccentricities (row maxima) are: vertex 1, $\max(0,1,-3,2,-4) = 2$; vertex 2, $\max(3,0,-4,1,-1) = 3$; vertex 3, $\max(7,4,0,5,3) = 7$; vertex 4, $\max(2,-1,-5,0,-2) = 2$; vertex 5, $\max(8,5,1,6,0) = 8$. The **diameter** is the largest, 8, attained by $\delta(5,1)$ along $5 \to 4 \to 1$. The **center** is not unique: both vertex 1 and vertex 4 have eccentricity 2. Ties are common in small graphs, and a tie-breaking rule — smallest index, or smallest row sum — must be stated rather than left to the implementation.

15.13 With $t_{ij}^{(0)} = 1$ exactly where $i = j$ or $w_{ij} < \infty$:

$$T^{(0)} = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}, \quad T^{(1)} = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}, \quad T^{(2)} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix},$$

$T^{(3)} = T^{(2)}$ (vertex 3's only out-edge goes to vertex 2, which adds nothing new), and $T^{(4)} = T^{(5)} = $ the all-ones matrix. The matrix first becomes all ones at round $k = 4$, and **vertex 4 is responsible**: it is the only vertex with an edge into vertex 1, so until routing through 4 is permitted, column 1 has zeros in rows 2, 3 and 5.

15.14 Listing 15.13 is one program that does all three.

Listing 15.13 The same triple loop over three semirings: shortest paths, widest paths, and shortest-path counts. Compile with `g++ -std=c++17 -O2 -o ex_semirings ex_semirings.cpp`.

```
// Chapter 15, Exercise solution: one triple loop, three different problems.
// The Floyd-Warshall skeleton does not care what "min" and "+" mean, only that
// the pair obeys the semiring laws. Change the pair and the problem changes.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

const long long INF = 1000000000LL;

void banner(const string& s) { cout << "\n" << s << "\n"; }

int main() {
    // ---------------------------------------------------------------- (min, +)
    // Shortest-path weights, on the five-vertex running example.
    {
        const int n = 5;
        vector<vector<long long>> D = {
            {  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}
        };
        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][j] = min(D[i][j], D[i][k] + D[k][j]);
        banner("(min,+) on the running example -- shortest-path weight:");
        for (int i = 0; i < n; ++i) {
            cout << "   ";
            for (int j = 0; j < n; ++j) cout << setw(5) << D[i][j];
            cout << '\n';
        }
    }
```

Listing 15.13 (Continued) *The same triple loop over three semirings: shortest paths, widest paths, and shortest-path counts. Compile with g++ -std=c++17 -O2 -o ex_semirings ex_semirings.cpp.*

```
// ---------------------------------------------------------------- (max, min)
// Widest path (maximum bottleneck). Edge values are capacities in liters per
// minute; b[i][j] is the largest flow a single route from i to j can carry.
{
    const int n = 5;
    const long long NONE = -1;
    // Capacities: 1->2 = 10, 2->4 = 5, 1->3 = 4, 3->4 = 8, 4->5 = 7.
    vector<vector<long long>> B(n, vector<long long>(n, NONE));
    vector<vector<long long>> cap = {{0,1,10},{1,3,5},{0,2,4},{2,3,8},{3,4,7}};
    for (int i = 0; i < n; ++i) B[i][i] = INF;      // a route of no edges is unlimited
    for (auto& c : cap) B[c[0]][c[1]] = c[2];
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (B[i][k] != NONE && B[k][j] != NONE)
                    B[i][j] = max(B[i][j], min(B[i][k], B[k][j]));
    banner("(max,min) -- widest path, bottleneck capacity:");
    for (int i = 0; i < n; ++i) {
        cout << "   ";
        for (int j = 0; j < n; ++j) {
            if (B[i][j] == NONE)     cout << setw(6) << "-";
            else if (B[i][j] == INF) cout << setw(6) << "inf";
            else                     cout << setw(6) << B[i][j];
        }
        cout << '\n';
    }
    cout << "   widest route from 1 to 4 carries " << B[0][3] << " l/min"
         << " (via 2, whose 5 beats the lower branch's 4)\n";
}

// ---------------------------------------------------------------- (+, x)
// How many distinct shortest paths join each pair. Two matrices march
// together: the weight D, and the count C of minimum-weight paths.
{
    const int n = 5;
    // 1->2, 1->3, 2->4, 3->4, 4->5 each of weight 1; plus a direct 1->5 of weight 3.
    vector<vector<long long>> E = {{0,1,1},{0,2,1},{1,3,1},{2,3,1},{3,4,1},{0,4,3}};
    vector<vector<long long>> D(n, vector<long long>(n, INF));
    vector<vector<long long>> C(n, vector<long long>(n, 0));
    for (int i = 0; i < n; ++i) { D[i][i] = 0; C[i][i] = 1; }
    for (auto& e : E) { D[e[0]][e[1]] = e[2]; C[e[0]][e[1]] = 1; }
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i) {
            if (i == k || D[i][k] >= INF) continue;
            for (int j = 0; j < n; ++j) {
                if (j == k || j == i || D[k][j] >= INF) continue;
```

Listing 15.13 (Continued) *The same triple loop over three semirings: shortest paths, widest paths, and shortest-path counts. Compile with g++ -std=c++17 -O2 -o ex_semirings ex_semirings.cpp.*

```
                    long long cand = D[i][k] + D[k][j];
                    if (cand < D[i][j])       { D[i][j] = cand; C[i][j] = C[i][k] *
C[k][j]; }
                    else if (cand == D[i][j]) { C[i][j] += C[i][k] * C[k][j]; }
                }
            }
        banner("(+,x) -- number of distinct shortest paths:");
        cout << "   weights:\n";
        for (int i = 0; i < n; ++i) {
            cout << "   ";
            for (int j = 0; j < n; ++j) {
                if (D[i][j] >= INF) cout << setw(5) << "inf"; else cout << setw(5) <<
D[i][j];
            }
            cout << '\n';
        }
        cout << "   counts:\n";
        for (int i = 0; i < n; ++i) {
            cout << "   ";
            for (int j = 0; j < n; ++j) cout << setw(5) << C[i][j];
            cout << '\n';
        }
        cout << "   delta(1,5) = " << D[0][4] << ", achieved by " << C[0][4]
             << " different paths\n";
    }
    return 0;
}
```

Its output:

```
(min,+) on the running example -- shortest-path weight:
       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

(max,min) -- widest path, bottleneck capacity:
      inf   10    4    5    5
        -  inf    -    5    5
        -    -  inf    8    7
        -    -    -  inf    7
        -    -    -    -  inf
   widest route from 1 to 4 carries 5 l/min (via 2, whose 5 beats the lower branch's 4)

(+,x) -- number of distinct shortest paths:
   weights:
       0    1    1    2    3
     inf    0  inf    1    2
     inf  inf    0    1    2
     inf  inf  inf    0    1
     inf  inf  inf  inf    0
   counts:
       1    1    1    2    3
       0    1    0    1    1
       0    0    1    1    1
       0    0    0    1    1
       0    0    0    0    1
   delta(1,5) = 3, achieved by 3 different paths
```

Checking by hand: the widest route from 1 to 4 goes either $1 \to 2 \to 4$, whose narrowest edge is $\min(10,5) = 5$, or $1 \to 3 \to 4$, whose narrowest is $\min(4,8) = 4$; the maximum is 5, as printed. The count $C_{15} = 3$ corresponds to $1 \to 2 \to 4 \to 5$, $1 \to 3 \to 4 \to 5$ and the direct $1 \to 5$, all of

weight 3; and $C_{14} = 2$ to $1 \to 2 \to 4$ and $1 \to 3 \to 4$. Note the two guards `i == k` and `j == k` in the counting loop: without them the degenerate candidate $d_{ii} + d_{ij} = d_{ij}$ would be counted as a second distinct path and the counts would double.

15.15 By Lemma 15.2 applied to a cycle p with $v_0 = v_k$, we get $\widehat{w}(p) = w(p) + h(v_0) - h(v_0) = w(p)$. Cycles keep their weight exactly. So a cycle is negative under $\widehat{w}$ if and only if it is negative under w, and reweighting can neither create nor destroy one. This matters for line 2 because Bellman–Ford is run on G' with the **original** weights w, before any reweighting — and G' has a negative cycle if and only if G does, since the added edges all leave s and none enters it, so no new cycle is created. If reweighting could destroy a negative cycle, a test performed at one stage would say nothing about the other, and the algorithm could proceed to run Dijkstra on a graph where no shortest path exists.

15.16 Because s **has no incoming edges**: to appear as an intermediate vertex of a path you must be both entered and left, and nothing enters s. Choosing $h(v) = \delta(u_0, v)$ for an existing vertex u_0 breaks because u_0 may not reach every vertex, so $h(v) = \infty$ for some v and Equation (15.13) becomes $\infty - \infty$, which is undefined. (When u_0 happens to reach every vertex, the choice actually works: Lemma 15.2 holds for *any* h, and the triangle inequality still makes every new weight non-negative. That u_0 has incoming edges and may lie inside other shortest paths is harmless.) The auxiliary vertex is simply the cheap, uniform way to guarantee that every $h(v)$ is finite.

15.17 From the output of **Listing 15.10**, the four zero-weight edges are (1,5), (2,4), (3,2) and (4,3). **The general statement.** $\widehat{w}(u, v) = w(u, v) + h(u) - h(v) = 0$ exactly when $h(v) = h(u) + w(u, v)$, that is, when $\delta(s, v) = \delta(s, u) + w(u, v)$. An edge satisfying this is called **tight**: reaching v by first taking a cheapest route to u and then crossing (u, v) is as cheap as reaching v at all, so (u, v) is the last edge of some cheapest route to v. Conversely, if (u, v) lies on a cheapest route from s, then by Theorem 13.1 the prefix ending at u is itself cheapest, so $\delta(s, v) = \delta(s, u) + w(u, v)$ and the edge is tight. **Checking the four**, with $h = (0, -1, -5, 0, -4)$ for vertices 1 to 5: (1,5) since $h(5) = -4 = 0 + (-4)$; (4,3) since $h(3) = -5 = 0 + (-5)$; (3,2) since $h(2) = -1 = -5 + 4$; (2,4) since $h(4) = 0 = -1 + 1$. All four are tight. **Do they form a tree?** No. Together with the zero-weight edges $s \to 1$ and $s \to 4$ of G', the tight edges contain the cycle $4 \to 3 \to 2 \to 4$, of weight $-5 + 4 + 1 = 0$. That is perfectly legal — the prohibition is on *negative* cycles, not zero ones — and it means the tight subgraph here is a shortest-path DAG containing a zero-weight cycle rather than a tree. A genuine shortest-paths tree can always be extracted by keeping exactly one incoming tight edge per vertex, but code that assumes the zero-weight edges are already a tree, and walks parent pointers without a visited check, will loop for ever on this graph.

15.18 Set $V^2 \lg V + VE = V^3$ and solve for E: dividing by V gives $V \lg V + E = V^2$, so $E = V^2 - V \lg V$. At $V = 600$ that predicts $E \approx 360{,}000 - 600 \times 9.2 \approx 354{,}500$ — that is, a density of about 98 percent, whereas the measurement in §15.9.2 put the crossover near 13 percent. The two differ because the asymptotic formulas discard constant factors, and the constants here are very different: Floyd–Warshall's inner statement is one addition and one comparison on contiguous memory, while Dijkstra's involves a priority-queue push, a pop, a cache-missing adjacency-list traversal and a stale-entry check. If we model the discrepancy as a single constant c multiplying Johnson's per-edge work, the crossover moves to $E \approx (V^2 - V \lg V)/c$; setting that equal to the measured 48,000 gives $c \approx 354{,}500/48{,}000 \approx 7.4$. A per-edge cost about seven times Floyd–Warshall's is a very plausible price for a heap push, a heap pop, a cache miss on the adjacency list and a stale-entry test. The lesson is the one drawn in §15.7.4: asymptotics choose the algorithm family, measurement chooses the member.

References

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

[2] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, Jun. 1962.

[3] S. Warshall, "A theorem on Boolean matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 11–12, Jan. 1962.

[4] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *Journal of the ACM*, vol. 24, no. 1, pp. 1–13, Jan. 1977.

[5] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[6] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[7] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[8] U. Zwick, "All pairs shortest paths using bridging sets and rectangular matrix multiplication," *Journal of the ACM*, vol. 49, no. 3, pp. 289–317, May 2002.

[9] Profitina, "AI Visibility (GEO) platform — product overview," Profitina Group, Surabaya, Indonesia, 2026. [Online]. Available: https://profitina.com

Chapter 16
Evaluation and Discussion IV: Shortest Paths, Amortized Analysis, and the Whole Book

This last chapter adds no new theory. It consolidates Chapters 13 to 15 — single-source shortest paths, amortized analysis and all-pairs shortest paths — and then steps back to integrate the whole book, by working problems. There are twenty-one of them, every one solved from nothing, with every intermediate number written down and every numerical claim checked by a complete C++17 program that has been compiled with `g++ -std=c++17 -O2 -Wall` and run. Five integrating problems open the chapter: fifteen true-or-false statements that range over the whole book, the longest increasing subsequence, a proof by induction about extended binary trees, a two-approximation for the traveling salesman, and a comparison of Bellman–Ford with Dijkstra. Seven problems then drill Chapters 13 to 15 one tool at a time: a Dijkstra trace with the whole distance-and-parent table after every iteration, Bellman–Ford pass by pass with a negative cycle detected and exhibited, shortest paths in a directed acyclic graph, the binary counter and the dynamic table each analyzed by the aggregate, accounting and potential methods with all the arithmetic shown, Floyd–Warshall with every matrix $D^{(0)}$ to $D^{(4)}$ computed cell by cell, and Johnson's reweighting carried out in full. Nine further problems revisit the rest of the book. Every problem ends with a *check yourself* variant and its answer. The chapter closes with forty quick self-check drills, an algorithm selection table that is the single page worth keeping from this book, a guided tour of six large territories the book did not enter, and a last word.

16.1 Where We Stand

16.1.1 What Chapters 13 to 15 gave us

Three chapters have given us a second toolkit, and every problem in the middle of this chapter reaches for one of its tools. It is worth naming them before we start.

From **Chapter 13** we have the single-source shortest-path problem: a weighted directed graph $G = (V, E)$ with weight function w, a source s, and the task of computing, for every vertex v, the shortest-path weight $\delta(s, v)$ together with a path that achieves it. We have the two arrays that every algorithm maintains — the estimate $d[v]$ and the predecessor $\pi[v]$ — and the one primitive that changes them, **relaxation** of an edge (u, v):

$$\text{if } d[v] > d[u] + w(u, v) \text{ then } d[v] \leftarrow d[u] + w(u, v), \;\; \pi[v] \leftarrow u. \tag{16.1}$$

We have the five properties of relaxation (upper bound, no path, convergence, path relaxation, predecessor subgraph), the triangle inequality $\delta(s, v) \leq \delta(s, u) + w(u, v)$, and three algorithms that differ only in the *order* in which they relax edges: Dijkstra's (nearest unfinished vertex first, non-negative weights only), Bellman–Ford (every edge, $|V| - 1$ times, negative weights allowed and negative cycles detected), and the DAG algorithm (vertices in topological order, one pass, any weights).

From **Chapter 14** we have **amortized analysis**: bounding the total cost of a *sequence* of n operations, so that one occasionally expensive operation is paid for by many cheap ones. We have its three methods — the **aggregate** method (compute the total, divide by n), the **accounting** method (charge each operation a fixed fee and keep a credit balance that must never go negative), and the **potential** method (define a potential Φ on the data structure and charge each operation its real cost plus the change in potential) — and three data structures on which they were practiced: the stack with `MULTIPOP`, the k-bit binary counter, and the dynamic table that doubles when full.

From **Chapter 15** we have the all-pairs problem and its three solutions: repeated "multiplication" of distance matrices, the **Floyd–Warshall** algorithm with its recurrence over *which vertices may be interior to a path*, and **Johnson's** algorithm, which reweights every edge with a potential h computed by one Bellman–Ford run so that $|V|$ runs of Dijkstra become legal. All three chapters follow Cormen et al. [1].

16.1.2 The whole book on one page

This chapter also integrates the whole book, so the whole toolkit is on the table. **Table 16.1** is the map. For each chapter it names the single idea that chapter exists to teach — the one sentence that, if you lost everything else, would let you rebuild the rest — and says where in this chapter you will see that idea at work.

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. $n \leq 10^5$ means $O(n\lg n)$ or better; $n \leq 500$ 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
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, $\|V\| - 1$ 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

16.1.3 How this chapter is organized

The chapter is built in four layers.

1. **A method** (Section 16.2): how to approach a problem that has several parts, or a proof that will not come out at the first attempt.
2. **Five integrating problems** (Sections 16.3 to 16.7), each worked at full length because each one draws on several chapters at once. Problem 16.4, the traveling-salesman approximation, is the showpiece: it concerns a subject the book never taught, and every ingredient of its solution is nevertheless already in your hands.
3. **Seven problems on Chapters 13 to 15** (Sections 16.8 to 16.10), each aimed at one tool, and nine **further worked problems** (Section 16.11) that revisit the rest of the book. Every one has the same shape, so that you always know where you are:
   - **Problem** — the statement.
   - **What it tests** — the tool, and where in the book it was introduced.

- **Solution** — step by step, with every arithmetic step shown.
- **Program** — a complete C++17 program, with the output it really printed.
- **Check yourself** — a variant for you to do alone, with its answer.

4. **Consolidation** (Sections 16.12 to 16.15): the problems compared side by side, forty self-check drills, the algorithm selection table, and where to go next.

Section 16.16 shows where the ideas of the whole book appear in a real product, and the chapter closes with the common mistakes, a summary and a closing word, fourteen exercises and their full solutions.

> **HOW TO USE THIS CHAPTER.** Read each problem statement, then cover the solution with a sheet of paper and try it yourself for ten minutes before reading on. A solution you have struggled with for ten minutes is understood in a way that a solution you have only read never is. The *check yourself* variants exist so that you can test that understanding on something you have not seen worked.

16.2 How to Approach a Multi-Part Problem or Proof

Several problems in this chapter have parts — *design, then analyze, then prove*; *describe, then write pseudocode, then illustrate* — and several ask for a proof. Both kinds reward the same discipline, and it is worth stating once, before the problems, rather than repeating it inside each.

16.2.1 Write the answer in a fixed order

Within any part, write in this order.

1. **The answer, in one sentence.** "The algorithm runs in $\Theta(n^2)$ time." "The statement is false." A reader — a colleague, a reviewer, or you yourself next month — looks for this sentence first. If it is buried in the fourth paragraph, it may never be found.
2. **The assumptions.** What exactly is the input? What does each word of the problem mean? Problem 16.2 turns entirely on the meaning of one word, *subsequence*, and an answer that does not state it may solve the wrong problem.
3. **The derivation or proof.**
4. **A worked example.** Never omit it. A small instance carried through by hand is the cheapest evidence in existence that you understood what you wrote, and it is also the fastest way to discover that you did not.

16.2.2 When a proof will not come out

A proof is not an atom. It is a sequence of claims, and each claim that is correct, clearly stated and relevant is progress. What destroys progress is not incompleteness but *incoherence*: three-quarters of a page of manipulation with no statement of what is being proved. So when a proof stalls, do this, in order.

1. **State the claim formally.** Write "Claim: $d(L) \leq 2\, d(L^*)$." It costs eight seconds and it tells you, and every later reader, where you are going.
2. **Name the technique and the quantity.** "By strong induction on n, the number of internal nodes." "By a potential function $= $ the number of 1 bits." This fixes what *kind* of object the proof is.
3. **Do the base case.** Base cases are almost always easy, and they are often where a false claim first shows itself.

4. **State the hypothesis precisely.** Not "assume it holds for smaller trees" but "assume $X(n') = n' + 1$ for every $0 \leq n' < n$."
5. **Write down the steps you *can* justify, and label the gap.** Literally write: "It remains to show that $d(W) \leq 2\,d(T)$." An honest, labeled gap is a professional act: it tells the next person — often you — exactly where to push. A disguised gap is a trap for everyone who reads the proof later.
6. **Check the claim on an example,** and say that it is an illustration, not a proof. It shows that you know what the claim *says*, and it very often suggests the missing step.

> **A STALLED PROOF IS REPAIRED BY STRUCTURE, NOT BY LENGTH.** Six clearly labeled lines — claim, technique, base, hypothesis, what follows, what is missing — are worth more than three pages of unlabeled algebra that stops in the middle. They show the shape of the argument even before the argument is complete, and the shape is usually what suggests the missing step.

16.2.3 Five habits worth keeping

- **Never write "obviously" or "clearly" in a proof.** If it is obvious, it takes one line: write the line.
- **Never state an O bound without saying which case it bounds.** "Quicksort is $O(n^2)$" and "quicksort is $O(n\lg n)$" are both true statements about different cases, and a sentence that does not say which is worth nothing. Statement (1) of Problem 16.1 turns on exactly this.
- **Give pseudocode before code.** A "design an algorithm" problem asks for an idea that a reader can check. A C++ listing is evidence that the idea works; it is not the idea.
- **Illustrate the algorithm, not the graph.** A picture of a shortest-path algorithm that does not show the estimates $d[v]$ changing is a picture of a graph.
- **Check every number twice** — once by hand and once by a program. Every table in this chapter was produced both ways, and they agree.

16.3 Problem 16.1 — Fifteen Statements, True or False

16.3.1 The problem

Problem 16.1. Mark each of the following statements True (T) or False (F), and justify each answer.

1. Quick-sort has worse asymptotic complexity than merge sort.
2. Binary search is $O(\log n)$.
3. Linear search in an unsorted array is $O(n)$, but in a sorted array is $O(\log n)$.
4. Linear search in a sorted linked list is $O(n)$.
5. If you only need to search for one value, linear search on an unsorted array is asymptotically better than sorting and doing binary search.
6. If all arc weights are unique, the minimum spanning tree is unique.
7. Binary search in an array requires the array to be sorted.
8. Insertion into an ordered list can be done in $O(\log n)$.
9. A good hash function distributes elements uniformly across the hash table.
10. With a good hash function and reasonable data, a search in a hash table can be done in $O(1)$.

11. If a code has asymptotic complexity $O(g(n))$, then at least $g(n)$ operations will be executed when it runs with parameter n.
12. Given good implementations for different algorithms for some processes, such as sorting, searching, or finding a minimum spanning tree, you should always choose the algorithm with the better asymptotic complexity.
13. It is not possible for the depth-first and breadth-first traversal of a graph to visit nodes in the same sequence if the graph contains more than two nodes.
14. The maximum number of nodes in a binary tree of height H ($H = 0$ for leaf nodes) is $2^{H+1} - 1$.
15. In a complete binary tree, only leaf nodes have no children.

16.3.2 What it tests

Not memory. Nine of the fifteen statements are true and six are false, and every false one is false for a *reason taught in this book*, not because of a misprint. The problem tests whether you can tell apart four pairs of ideas that are habitually merged:

- an **upper** bound and an **exact** count (statements 11, 12);
- an algorithm's **worst** case and its **average** case (statement 1);
- what a data structure's **shape** permits and what a particular **algorithm** on it does (statements 3, 4, 8);
- what is **guaranteed** and what is **typical** (statements 9, 10, 12).

Work through the fifteen with those four pairs in mind and the problem takes fifteen minutes. Work through them by recall and it takes an hour, and statement (11) will still go wrong.

16.3.3 The answers

Table 16.2 gives the fifteen answers together with the one-line reason for each. The subsections after it derive the ones that deserve a derivation.

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 ⇒ 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 ⇒ $O(1)$ 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
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*

16.3.4 Statement (1) — the one that needs a sentence of qualification

"Quick-sort has worse asymptotic complexity than merge sort."

The answer is **True**, but only under a reading that you should make explicit.

Merge sort runs in $\Theta(n\lg n)$ in the best case, the average case and the worst case; the split is by position, so the data cannot change the recurrence. Quicksort's recurrence depends on the pivot. With a balanced split, $T(n) = 2T(n/2) + \Theta(n) = \Theta(n\lg n)$. With the worst possible split — one element on one side, $n-1$ on the other, which a last-element pivot produces on an already-sorted array — the recurrence is

$$T(n) = T(n-1) + \Theta(n) = \Theta(n^2).$$

So:

- **worst case:** quicksort $\Theta(n^2)$, merge sort $\Theta(n\lg n)$. Quicksort is worse. **True.**
- **average case:** both $\Theta(n\lg n)$. Neither is worse. The statement would be **false**.

When a statement is true under one standard reading and false under another, the substance is in the qualification. Write: **"True, reading 'asymptotic complexity' as worst-case: $\Theta(n^2)$ against $\Theta(n\lg n)$. In the average case both are $\Theta(n\lg n)$ and the statement would be false."** One extra clause converts a guess into a demonstration.

THE MOST COMMON MISTAKE ON STATEMENT (1). Many readers answer *False* because they remember that quicksort is faster than merge sort in practice. It is — smaller constants, better locality, sorting in place. But "faster in practice" and "worse asymptotic complexity" are both true at once, and the statement asks about the second. Practice belongs in statement (12), where it is the whole point.

16.3.5 Statements (3), (4) and (7) — sortedness and the algorithm that uses it

These three are one idea seen from three sides: **sorting the data changes what an algorithm *can* do; it does not change what a particular algorithm *does*.**

(3) "Linear search in an unsorted array is $O(n)$, but in a sorted array is $O(\log n)$." **False.** Linear search examines $A[1], A[2], \ldots$ in order until it finds the key or runs out. On a sorted array it may stop early when it passes the key, but the worst case — the key is the last element, or is larger than every element — still costs n comparisons. Linear search is $\Theta(n)$ on sorted and unsorted input alike. What *is* $O(\lg n)$ on a sorted array is binary search, which is a different algorithm.

(4) "Linear search in a sorted linked list is $O(n)$." **True**, and for a reason stronger than in (3). In a linked list you cannot reach the middle element without walking to it. Binary search is not merely unhelpful here; it is unavailable, because the halving step needs $\Theta(1)$ access to position $\lfloor (lo + hi)/2 \rfloor$, and a list gives $\Theta(n)$.

(7) "Binary search in an array requires the array to be sorted." **True.** The algorithm's whole justification is the discard step: after comparing the key with $A[\text{mid}]$ it throws away half the array. That throw-away is only sound if sortedness guarantees the key cannot be in the discarded half. Remove the ordering and the discard is a guess.

16.3.6 Statement (5) — when not to sort

"If you only need to search for one value, linear search on an unsorted array is asymptotically better than sorting and doing binary search." **True.**

The arithmetic is short, and it is the arithmetic that makes the answer convincing. Linear search costs $\Theta(n)$. The alternative costs

$$\underbrace{\Omega(n \lg n)}_{\text{the sort}} + \underbrace{O(\lg n)}_{\text{the search}} = \Omega(n \lg n),$$

and n grows more slowly than $n\lg n$, so linear search wins asymptotically. The lower bound $\Omega(n\lg n)$ on the sort is not a property of one sorting algorithm; it is the comparison-sort lower bound of Chapter 5, so no cleverness recovers the difference.

The natural follow-up: *at how many searches does sorting pay for itself?* With k searches, linear costs $\Theta(kn)$ and sort-then-search costs $\Theta(n\lg n + k\lg n)$. These cross when $kn \approx n\lg n$, that is at $k \approx \lg n$. For a million elements, $\lg n \approx 20$: sort if you expect more than about twenty lookups, scan if fewer.

16.3.7 Statement (8) — the trap with two halves

"Insertion into an ordered list can be done in $O(\log n)$." **False**, and it is the statement of the fifteen most often answered wrongly.

Insertion has two parts.

- **Find where the new element goes.** In a sorted *array*, binary search does this in $O(\lg n)$.

- **Put it there.** In a sorted array, every element after the insertion point must move one place right: $\Theta(n)$ in the worst case.

$O(\lg n) + \Theta(n) = \Theta(n)$. The cheap half does not rescue the expensive half.

In a sorted *linked list* the two halves swap roles: inserting a node once you hold a pointer costs $\Theta(1)$, but finding the position costs $\Theta(n)$ because you must walk. Either way the total is $\Theta(n)$. The statement is false for both list representations, which is why it is false without qualification.

What *is* $O(\lg n)$? Insertion into a balanced search tree — an AVL tree or a red–black tree, Chapter 6 — or into a skip list. Those structures exist precisely because ordered arrays and ordered lists cannot do it.

16.3.8 Statements (9), (10) and (12) — guaranteed against typical

(9) "A good hash function distributes elements uniformly across the hash table." **True** — that is what *good* means. The assumption of *simple uniform hashing* says each key is equally likely to land in any of the m slots, independently of the others.

(10) "With a good hash function and reasonable data, a search in a hash table can be done in $O(1)$." **True.** Under simple uniform hashing with chaining, the expected search time is $\Theta(1 + \alpha)$, where $\alpha = n/m$ is the load factor. If the table is resized so that α stays below a constant — the dynamic table of Problem 16.10, applied to a hash table — the expected cost is $\Theta(1)$. Note the careful words: *expected*, and *can be*. The worst case, when every key collides, is $\Theta(n)$ — and the statement, saying "can be done", is compatible with that.

(12) "You should always choose the algorithm with the better asymptotic complexity." **False**, and the word doing the work is *always*. Four standard counter-examples, any one of which settles it:

1. **Constants.** Insertion sort is $\Theta(n^2)$ and merge sort is $\Theta(n \lg n)$, yet every industrial sort switches to insertion sort below about sixteen elements, because the constant is smaller and the data fits in cache. Chapter 5.
2. **Memory.** Merge sort needs $\Theta(n)$ extra space; heapsort needs $\Theta(1)$. On a device with 64 KB of RAM, heapsort wins even though the asymptotics tie.
3. **Input size.** Asymptotic means "for large enough n". If n is always 30, the asymptotics are simply not about your problem.
4. **Other properties.** Merge sort is stable; quicksort and heapsort are not. If the specification needs stability, the asymptotics do not get a vote.

For the minimum spanning tree case that the statement names explicitly: Kruskal is $O(E \lg E)$ and Prim with a Fibonacci heap is $O(E + V \lg V)$, which is asymptotically better — yet Kruskal is often faster on sparse graphs and is far simpler to implement correctly, and a Fibonacci heap's constants are notoriously bad. The better bound is not the better choice.

16.3.9 Statement (6) — why distinct weights force a unique MST

"If all arc weights are unique, the minimum spanning tree is unique." **True.** This deserves a real proof; it is the only statement of the fifteen that does.

Suppose all edge weights are distinct and, for contradiction, that T_1 and T_2 are two different minimum spanning trees. Since they differ, some edge lies in one and not the other. Among all edges in the symmetric difference $(T_1 \setminus T_2) \cup (T_2 \setminus T_1)$, let e be the one of **smallest weight**; weights are distinct, so e is unique. Without loss of generality $e \in T_1$ and $e \notin T_2$.

Add e to T_2. A tree plus one edge contains exactly one cycle, call it C, and C passes through e. Not every edge of C can be in T_1, because T_1 is acyclic and contains e. So there is an edge $f \in C$ with $f \in T_2$ and $f \notin T_1$ — that is, f is also in the symmetric difference. Because e was the lightest edge of the symmetric difference and weights are distinct, $w(e) < w(f)$.

Now form $T_3 = T_2 \cup \{e\} \setminus \{f\}$. Removing an edge of the unique cycle leaves a spanning tree, so T_3 is a spanning tree, and

$$w(T_3) = w(T_2) + w(e) - w(f) < w(T_2),$$

contradicting the minimality of T_2. Hence no two distinct minimum spanning trees exist. ■

> **THE CONVERSE IS FALSE, AND KNOWING THAT IS PART OF THE ANSWER.** Distinct weights *force* uniqueness, but ties do not force multiplicity. The six-point graph of Section 16.6.7 has three edges of weight 5 and still exactly one minimum spanning tree — a brute-force enumeration of all $\binom{15}{5}$ five-edge subsets finds exactly one of minimum weight. Ties make a second MST *possible*, not *certain*.

16.3.10 Statements (11), (14) and (15) — bounds, counts and definitions

(11) "If a code has asymptotic complexity $O(g(n))$, then at least $g(n)$ operations will be executed." **False**, and it is the statement most often answered wrongly by readers who are otherwise fluent.

$f(n) = O(g(n))$ means: there exist $c > 0$ and n_0 such that $0 \le f(n) \le c\, g(n)$ for all $n \ge n_0$. It is an **upper** bound. It says the code does *at most* about $g(n)$ operations, not at least. Concretely, a program that prints "hello" once and stops is $O(n^2)$ — the definition is satisfied with $c = 1$ — and it executes a constant number of operations, not n^2 of them. The statement you may safely make about *at least* is Ω: $f(n) = \Omega(g(n))$ means $f(n) \ge c\, g(n)$ eventually.

(14) "The maximum number of nodes in a binary tree of height H (with $H = 0$ for a leaf) is $2^{H+1} - 1$." **True.** Level i of a binary tree holds at most 2^i nodes — level 0 holds the root, and each node has at most two children, so each level at most doubles. A tree of height H has levels 0 through H, so at most

$$\sum_{i=0}^{H} 2^i = 2^{H+1} - 1$$

nodes, by the geometric sum of Chapter 3. Check the small cases: $H = 0$ gives $2^1 - 1 = 1$, a single node; $H = 1$ gives $2^2 - 1 = 3$, a root and two children; $H = 2$ gives 7. Correct.

Note the convention. This book sets a leaf's height to 0; some texts set it to 1, and then the bound reads $2^H - 1$. If a problem does not state the convention, state the one you are using. Here the statement gives it, in brackets, and the bracket is the whole reason the exponent is $H + 1$.

(15) "In a complete binary tree, only leaf nodes have no children." **True**, and it is true rather more cheaply than most readers expect: a *leaf* is **defined** as a node with no children, so "only leaves have no children" holds in every tree whatsoever, complete or not. Some readers look for a trap here and invent one. There is none. Answer T and move on — that is thirty seconds saved for statement (11), which needs them.

16.3.11 Statement (13) — the one you can refute in five seconds

"It is not possible for the depth-first and breadth-first traversal of a graph to visit nodes in the same sequence if the graph contains more than two nodes." **False.**

Two refutations, either of which is enough:

- **A path.** Take vertices $1 - 2 - 3$ and start at 1. Breadth-first visits 1, then 2, then 3. Depth-first visits 1, goes deep to 2, goes deep to 3. Same sequence, three nodes.
- **A star.** Take a center c joined to a, b, d, and start at c. Breadth-first: c, a, b, d. Depth-first: c, down to a, back up, down to b, back up, down to d — c, a, b, d. Same sequence, four nodes.

Far from being impossible, agreement is *common*. **Listing 16.1** enumerates every graph on four labeled vertices in which vertex 0 can reach all the others, and counts how often the two orders coincide.

Listing 16.1 An exhaustive refutation of statement (13): every graph on four labeled vertices is built, and the depth-first and breadth-first orders from vertex 0 are compared.

```
// Problem 16.1, statement (13), claims that depth-first and breadth-first
// order can never agree on a graph of more than two vertices. This program
// enumerates every graph on four labeled vertices and counts the ones where
// they do agree, starting from vertex 0 and always scanning neighbors in
// increasing order.
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

const int n = 4;
bool adj[n][n];

void dfs(int u, vector<bool>& seen, vector<int>& out) {
    seen[u] = true; out.push_back(u);
    for (int v = 0; v < n; ++v) if (adj[u][v] && !seen[v]) dfs(v, seen, out);
}
vector<int> bfs(int s) {
    vector<bool> seen(n, false); vector<int> out;
    queue<int> Q; Q.push(s); seen[s] = true;
    while (!Q.empty()) {
        int u = Q.front(); Q.pop(); out.push_back(u);
        for (int v = 0; v < n; ++v) if (adj[u][v] && !seen[v]) { seen[v] = true; Q.push(v);
}
    }
    return out;
}

int main() {
    vector<pair<int,int>> pairs;
    for (int i = 0; i < n; ++i) for (int j = i + 1; j < n; ++j) pairs.push_back({i, j});
    int m = (int)pairs.size(), agree = 0, connected = 0, shown = 0;

    for (int mask = 0; mask < (1 << m); ++mask) {
        for (int i = 0; i < n; ++i) for (int j = 0; j < n; ++j) adj[i][j] = false;
        for (int k = 0; k < m; ++k) if (mask >> k & 1)
            adj[pairs[k].first][pairs[k].second] = adj[pairs[k].second][pairs[k].first] =
true;
```

Listing 16.1 (Continued) *An exhaustive refutation of statement (13): every graph on four labeled vertices is built, and the depth-first and breadth-first orders from vertex 0 are compared.*

```
        vector<bool> seen(n, false); vector<int> D;
        dfs(0, seen, D);
        if ((int)D.size() != n) continue;                // reach all four, or skip
        ++connected;
        vector<int> B = bfs(0);
        if (D == B) {
            ++agree;
            if (shown < 3) {
                ++shown;
                cout << "edges:";
                for (int k = 0; k < m; ++k) if (mask >> k & 1)
                    cout << " " << pairs[k].first << "-" << pairs[k].second;
                cout << "   order:";
                for (int v : D) cout << " " << v;
                cout << "\n";
            }
        }
    }
    cout << "\nconnected graphs on 4 labeled vertices reachable from 0: " << connected
         << "\nof those, DFS order = BFS order in " << agree << " of them\n";
    return 0;
}
```

Compiled with `g++ -std=c++17 -O2 -Wall` and run, it prints:

```
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
```

Twenty-six out of thirty-eight. The statement is not merely false; it is false in the majority of cases. One sentence and one three-vertex path is a complete answer — but it is satisfying to know how wrong the claim is.

16.3.12 The mistake most often made on this problem

Answering the statement you expected instead of the statement that is written. Statement (3) is read as "binary search in a sorted array is $O(\log n)$" — which is true — instead of what it says, which is about *linear* search. Statement (11) is read as "$O(g(n))$ means about $g(n)$ operations". Statement (12) is read without the word *always*.

The defense costs thirty seconds: **underline the verb and the qualifier in every statement before you answer it.** In (3) the verb-phrase is *linear search*. In (11) the qualifier is *at least*. In (12) it is *always*. In (13) it is *not possible*. Four underlines, and four answers that would otherwise have been coin-flips.

Check yourself. Decide, with a one-line reason: (a) "Dijkstra's algorithm runs in $O(E \lg V)$ with a binary heap, so it is always faster than Bellman–Ford." (b) "Relaxing an edge can never increase $d[v]$." (c) "The amortized cost of an operation is never smaller than its cheapest actual cost."

Answers. (a) **False** — the bound is right, but Dijkstra is not applicable at all when some weight is negative, and *always* ignores that; statement (12) again. (b) **True** — relaxation assigns $d[u] + w(u, v)$ only when it is *smaller* than $d[v]$ (Equation (16.1)). (c) **False** — in the potential analysis of the stack, `POP` has actual cost 1 and amortized cost 0; the amortized cost of a cheap operation can be below its actual cost because earlier operations prepaid it.

16.4 Problem 16.2 — The Longest Monotonically Increasing Subsequence

16.4.1 The problem

Problem 16.2. Design a polynomial-time algorithm to find the longest monotonically increasing subsequence in a sequence of n unique integers; each integer appears once in the input. Give your assumptions, pseudocode, and a time-complexity analysis.

16.4.2 What it tests

Three things, in the order the problem lists them.

Assumptions test whether you can state what a *subsequence* is without being told. This is the whole difficulty of the problem hiding in one word. A **subsequence** is obtained by deleting zero or more elements and keeping the rest in their original order; it need not be contiguous. A **substring** or **subarray** is contiguous. Solve the contiguous problem and you will produce a correct $O(n)$ algorithm for a problem nobody asked.

Pseudocode tests whether you can express a dynamic program. Chapter 7 gave the four steps: characterize the structure of an optimal solution, write a recurrence for its value, compute the value bottom-up, construct the solution from the recorded choices. The fourth step — reconstruction — is the one most often forgotten, because it is easy to compute the *length* and forget that the problem asks for the *subsequence*.

Time-complexity analysis tests whether you can count two nested loops honestly, and whether you notice that the reconstruction is free.

The word *polynomial* in the problem is a floor, not a target. $O(n^2)$ is polynomial and is the natural answer. Section 16.4.9 gives $O(n\lg n)$, which is better, and which shows what it means to go past the first correct answer.

16.4.3 Assumptions, stated properly

Write these five lines; each one closes a door through which a wrong answer could walk in.

1. The input is $S[1..n]$, an array of n integers.
2. All n integers are **distinct** — the problem says so. This matters: with duplicates you would have to decide between *strictly* increasing and *non-decreasing*, and the two give different answers. Distinctness makes the problem unambiguous.
3. *Increasing* means **strictly** increasing: $S[i_1] < S[i_2] < \cdots < S[i_k]$.
4. A **subsequence** keeps relative order but need not be contiguous: indices $1 \le i_1 < i_2 < \cdots < i_k \le n$.
5. The empty sequence and every single element are trivially increasing, so the answer has length at least 1 whenever $n \ge 1$.

16.4.4 Deriving the recurrence

The obstacle to a direct recursion is that "the longest increasing subsequence of $S[1..i]$" is not a quantity you can extend. Knowing that $S[1..5]$ has a longest increasing subsequence of length 3 tells you nothing about whether $S[6]$ can be appended, because you do not know what that subsequence *ends with*.

The repair is the standard one for this family of problems: **strengthen the subproblem until it composes.** Define

$$L[i] \\ = \text{the length of the longest increasing subsequence of } S[1..i] \text{ that ends exa}$$

The words *ends exactly at* $S[i]$ are the entire insight. With them, composition works.

Optimal substructure. Let T be a longest increasing subsequence ending at $S[i]$, and suppose it has length ≥ 2. Let $S[j]$ be the element immediately before $S[i]$ in T. Then $j < i$ and $S[j] < S[i]$, and the part of T up to $S[j]$ is an increasing subsequence of $S[1..j]$ ending at $S[j]$. Moreover it must be a *longest* such subsequence: if there were a longer one ending at $S[j]$, we could swap it in and get a longer subsequence ending at $S[i]$, contradicting the choice of T. That is the cut-and-paste argument of Chapter 7, applied here.

So the length of T is $L[j] + 1$, and since j could be any index with $j < i$ and $S[j] < S[i]$, we take the best:

$$L[i] = \begin{cases} \max\limits_{j<i \,\wedge\, S[j]<S[i]} (L[j]+1) & \text{if such a } j \text{ exists,} \\ 1 & \text{otherwise.} \end{cases} \tag{16.2}$$

The answer. The longest increasing subsequence of the whole array ends *somewhere*, so its length is $\max_{1 \leq i \leq n} L[i]$.

Reconstruction. Store $P[i]$, the index j that achieved the maximum for $L[i]$, or 0 if $L[i] = 1$. Following P from the maximizing k back to 0 lists the subsequence in reverse.

The loop invariant that makes the bottom-up computation of Equation (16.2) correct: *at the start of the iteration for index i,* $L[1..i-1]$ *and* $P[1..i-1]$ *hold the correct values for their subproblems.* Initialization is vacuous at $i = 1$; maintenance is the recurrence above, which reads only entries $j < i$; termination at $i = n + 1$ gives the whole table. That is the three-part obligation of Chapter 2, discharged in three lines.

16.4.5 The pseudocode, read line by line

Pseudocode 16.1 is the algorithm, with the backtracking that turns a table of lengths into an actual subsequence.

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]]
```

Two details in line 5 deserve a sentence each, because both look like slips and neither is.

Why `L[j] >= L[i]` and not `L[j] + 1 > L[i]`? They are the same test. Adding one to both sides of $L[j] \geq L[i]$ gives $L[j] + 1 \geq L[i] + 1 > L[i]$, and conversely $L[j] + 1 > L[i]$ over integers means $L[j] \geq L[i]$. The first form saves an addition. Either is correct; it is worth checking for yourself that they agree.

Why does it keep the *last* qualifying j rather than the first? Because the test is $\geq$ rather than $>$, a later j with an equal $L[j]$ overwrites an earlier one. This changes *which* longest subsequence is reported, not its length. Both are valid answers.

A tempting slip, and why it is harmless here but should still be fixed. It is tempting to write the recurrence with the range $1 \leq j \leq i$, which allows $j = i$. It does no damage — the guard $S[j] < S[i]$ is false when $j = i$, since no number is less than itself — but the correct range is $1 \leq j < i$, and that is what line 4 of **Pseudocode 16.1** actually implements. A recurrence and its code should say the same thing; when they differ harmlessly today, they tend to differ harmfully after the next edit.

16.4.6 A complete worked example, by hand

Take

$$S = \langle 9,\ 2,\ 5,\ 3,\ 7,\ 11,\ 8,\ 10,\ 13,\ 6 \rangle, \qquad n = 10.$$

Work the outer loop one index at a time. At each step we scan $j = 1, \ldots, i - 1$ and keep the best.

- $i = 1, S[1] = 9$. No j. $L[1] = 1, P[1] = 0$.
- $i = 2, S[2] = 2$. $j = 1$: is $9 < 2$? No. $L[2] = 1, P[2] = 0$.
- $i = 3, S[3] = 5$. $j = 1$: $9 < 5$? No. $j = 2$: $2 < 5$ yes, and $L[2] = 1 \geq L[3] = 1$, so $L[3] \leftarrow 1 + 1 = 2$, $P[3] \leftarrow 2$.
- $i = 4$, $S[4] = 3$. $j = 2$: $2 < 3$ yes, $L[2] = 1 \geq 1$, so $L[4] \leftarrow 2$, $P[4] \leftarrow 2$. $j = 3$: $5 < 3$? No. Result $L[4] = 2, P[4] = 2$.
- $i = 5$, $S[5] = 7$. $j = 2$: $2 < 7$, $L[2] = 1 \geq 1$, so $L[5] \leftarrow 2$, $P[5] \leftarrow 2$. $j = 3$: $5 < 7$, $L[3] = 2 \geq 2$, so $L[5] \leftarrow 3$, $P[5] \leftarrow 3$. $j = 4$: $3 < 7$, but $L[4] = 2 \geq 3$ is false. Result $L[5] = 3, P[5] = 3$.
- $i = 6$, $S[6] = 11$. $j = 1$: $9 < 11$, $L[1] = 1 \geq 1$, $L[6] \leftarrow 2$, $P[6] \leftarrow 1$. $j = 3$: $5 < 11$, $L[3] = 2 \geq 2$, $L[6] \leftarrow 3, P[6] \leftarrow 3$. $j = 5$: $7 < 11, L[5] = 3 \geq 3, L[6] \leftarrow 4, P[6] \leftarrow 5$. Result $L[6] = 4, P[6] = 5$.
- $i = 7$, $S[7] = 8$. The qualifying j are 2,3,4,5; the best is $j = 5$ with $L[5] = 3$. Result $L[7] = 4$, $P[7] = 5$.

- $i = 8$, $S[8] = 10$. Qualifying j are 1,2,3,4,5,7; the last to improve is $j = 7$ with $L[7] = 4$. Result $L[8] = 5$, $P[8] = 7$.
- $i = 9$, $S[9] = 13$. The last improvement is $j = 8$ with $L[8] = 5$. Result $L[9] = 6$, $P[9] = 8$.
- $i = 10$, $S[10] = 6$. Qualifying j are 2,3,4; the best is $j = 3$ with $L[3] = 2$. Result $L[10] = 3$, $P[10] = 3$.

Table 16.3 collects the two arrays.

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

The largest entry is $L[9] = 6$, so $k = 9$. Backtrack:

$$j = 9 \to S[9] = 13; \quad j = P[9] = 8 \to 10; \quad j = P[8] = 7 \to 8; \quad j = P[7] = 5 \to 7; \quad j = P[5] = 3 \to 5; \quad j = P[3] = 2 \to 2; \quad j = P[2] = 0 \text{ stop.}$$

Reading the collected values backward:

$$\langle 2,\ 5,\ 7,\ 8,\ 10,\ 13 \rangle, \qquad \text{length } 6.$$

Check it: the values increase, and they occur at positions $2 < 3 < 5 < 7 < 8 < 9$, which increase too. It is a valid increasing subsequence of length 6.

16.4.7 The time complexity, counted honestly

Lines 1–7 of **Pseudocode 16.1** are two nested loops. The outer runs n times; for a given i the inner runs $i - 1$ times. The body is $\Theta(1)$. The total number of inner iterations is

$$\sum_{i=1}^{n}(i-1) = \sum_{m=0}^{n-1} m = \frac{(n-1)n}{2} = \frac{n^2 - n}{2},$$

which is $\Theta(n^2)$. Line 8 and the backtrack of lines 9–14 each make one pass — the backtrack visits at most n indices because $P[j] < j$ strictly, so the chain cannot revisit an index — giving $\Theta(n)$ each. Therefore

$$T(n) = \Theta(n^2) + \Theta(n) + \Theta(n) = \Theta(n^2).$$

Space is $\Theta(n)$ for L, P and T together. Saying $O(n^2)$ is correct; $\Theta(n^2)$ is sharper and equally correct, since the inner loop always runs to completion — there is no early exit, so no best case is faster.

Is n^2 fast enough? Chapter 4's rule says read the constraint as a budget. At $n = 10^4$ we do about 5×10^7 inner iterations, well under a second. At $n = 10^6$ we would do 5×10^{11} — hours. That is the practical reason to know Section 16.4.9.

16.4.8 The program

Listing 16.2 is **Pseudocode 16.1** as a complete C++ program, printing the whole table so it can be checked against **Table 16.3** entry by entry.

Listing 16.2 The $O(n^2)$ algorithm of Problem 16.2 in C++, printing the L and P tables.

```
// 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;
}
```

Compiled with `g++ -std=c++17 -O2 -Wall` and run:

```
 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
```

Every entry agrees with the hand computation of **Table 16.3**.

16.4.9 A second method: $O(n\lg n)$

There is a better algorithm, and it is worth half a page, because it shows how a second idea — a sorted auxiliary array — removes a whole factor of n.

Keep an array `tail`, where `tail[k]` is **the smallest value that can end an increasing subsequence of length $k+1$ among the elements seen so far.** Two facts make this work:

1. **`tail` is sorted.** If a subsequence of length $k+1$ can end at value v, then its first k elements form a subsequence of length k ending at something smaller than v; so `tail[k-1] < tail[k]`.
2. **Each new element replaces exactly one entry.** Given $S[i]$, find the first position k with `tail[k] >= S[i]` — a binary search, because `tail` is sorted. If no such position exists, $S[i]$ extends the longest run and is appended. Otherwise `tail[k]` is replaced by $S[i]$, which is smaller and therefore at least as useful for the future, and never harms a shorter or longer entry.

The length of `tail` at the end is the length of the longest increasing subsequence. To recover the subsequence itself, record for each element the index of the element that preceded it — that is `tailIdx[k-1]` at the moment of insertion — and follow the chain back.

Cost: n binary searches on an array of length at most n, so $O(n\lg n)$, with $\Theta(n)$ space. **Listing 16.3** implements it.

Listing 16.3 The $O(n\lg n)$ method for Problem 16.2, printing the `tail` array after every element.

```
// 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;
}
```

Its output:

```
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
```

Two things to notice.

First, **the answer is the same length but a different subsequence**: ⟨2,3,7,8,10,13⟩ against the dynamic program's ⟨2,5,7,8,10,13⟩. Both are longest increasing subsequences. The longest increasing subsequence is not unique, and a problem that asks for "the" longest is asking for *a* longest. Saying so shows that you understand what you computed.

Second, **`tail` is not itself a subsequence of S.** After the last element, `tail` reads 2,3,6,8,10,13 — but 6 occurs at position 10, after 8, 10 and 13. The array is a table of best-known endings, not an answer. Printing `tail` as the answer is the classic error with this method; the reconstruction through `parent` is what produces a genuine subsequence.

16.4.10 The mistakes most often made on this problem

The first is solving the **contiguous** problem. "Longest increasing run" for the same input is ⟨2,5⟩ or ⟨3,7,11⟩ — length 3, not 6 — and it is computable in one pass. A clean $O(n)$ algorithm with a correct analysis is still wrong if it answers a different problem.

The defense is the fourth of the five assumptions in Section 16.4.3. Writing "a subsequence need not be contiguous" at the top of the answer forces you to notice, before you design anything, which problem you are solving.

The second is computing $\max_i L[i]$ and stopping. The problem says *find the longest monotonically increasing subsequence*, not *find its length*. Without the P array and the backtrack you have answered a smaller problem.

Check yourself. Run **Pseudocode 16.1** by hand on $S = \langle 3,1,8,2,5,9,4 \rangle$. Give L, P and the subsequence.

Answer. $L = \langle 1,1,2,2,3,4,3 \rangle$ and $P = \langle 0,0,1,2,4,5,4 \rangle$. (At $i = 3$, $j = 1$ sets $L[3] = 2$; then $j = 2$ has $L[2] = 1 \geq 2$ false, so $P[3]$ stays 1 — the $\geq$ test prefers a later j only when it is at least as good. At $i = 5$, $S[5] = 5$: $j = 4$ with $S[4] = 2$ and $L[4] = 2$ wins.) The maximum is $L[6] = 4$; backtracking $6 \to 5 \to 4 \to 2$ gives ⟨1,2,5,9⟩.

16.5 Problem 16.3 — Extended Binary Trees, Proved by Induction

16.5.1 The problem

Binary trees are used for search: the search starts at the root and proceeds to the left or the right child depending on a comparison. Sometimes the search stops at a node of the tree. Sometimes it tries to cross a null link to a child that does not exist; the search is then said to *fall out* of the tree. Falling out loses information — you no longer know *where* it happened — and that knowledge is

often useful. So, given any binary tree, form its **extended binary tree** by replacing each missing child (each null pointer) with a special leaf, an **external node**. The original nodes are then called **internal nodes**. If the original tree is empty, the extended tree consists of a single external node. In an extended binary tree every internal node has exactly two children and every external node has none.

Figure 16.1 shows an example, with external nodes drawn as squares and internal nodes as circles.

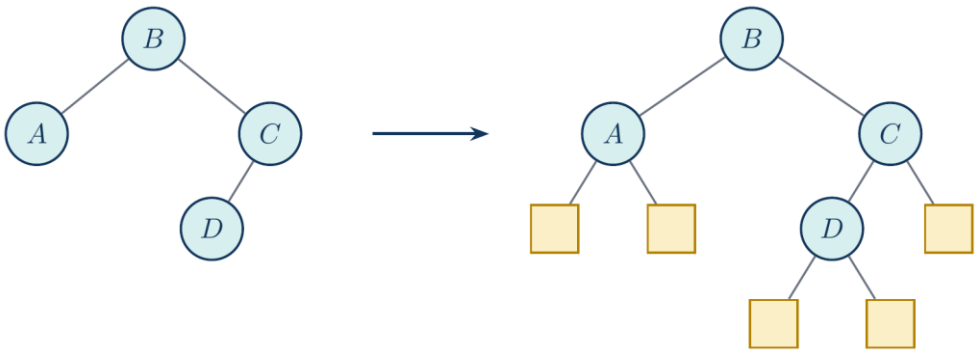


Figure 16.1 An ordinary binary tree with four nodes (left) and the extended binary tree obtained from it (right). Circles are internal nodes; squares are the external nodes that replace the null pointers. Four internal nodes, five external ones.

Problem 16.3. Let n denote the number of internal nodes in an extended binary tree. Prove by induction that an extended binary tree with n internal nodes has $n + 1$ external nodes.

Count the squares on the right of **Figure 16.1**: node A contributes two, node C one, node D two — five external nodes for four internal ones. That is the claim, on one instance.

16.5.2 What it tests

Whether you can write an induction **on the right quantity**, with the hypothesis stated precisely.

The trap is to induct on the height of the tree. Height induction fails here, or at least becomes painful, because the two subtrees of a node need not have the same height and the inductive hypothesis has to be strong enough to cover both. The right quantity is the **number of internal nodes**, and the right form is **strong** induction: the hypothesis must cover *every* smaller size, because a tree with n internal nodes splits into subtrees of sizes k and $n - 1 - k$, and k can be anything from 0 to $n - 1$.

This is a proof structure worth owning outright. Whenever you are asked to prove a statement about binary trees by induction, the same general structure applies — split at the root, apply the hypothesis to both subtrees, add — and Section 16.11 uses the same skeleton again.

16.5.3 The proof

Let $X(n)$ denote the number of external nodes in an extended binary tree whose number of **internal** nodes is n. We prove $X(n) = n + 1$ for every integer $n \geq 0$, by strong induction on n.

Base case, $n = 0$. A binary tree with no nodes is empty. By definition, extending the empty tree produces a single external node. Hence $X(0) = 1 = 0 + 1$, as required.

Inductive hypothesis. Let $n \geq 1$ and assume that $X(n') = n' + 1$ for every n' with $0 \leq n' < n$.

Inductive step. Consider an extended binary tree with $n \geq 1$ internal nodes. Because $n \geq 1$, the original tree is non-empty and so has a root, which is an internal node. Remove the root; the remaining $n - 1$ internal nodes are divided between the left subtree and the right subtree. Let k be the number of internal nodes in the left subtree, so the right subtree has $(n - 1) - k$ internal nodes. Both numbers lie in $[0, n - 1]$, and both are strictly less than n, so the inductive hypothesis applies to each.

Each subtree is itself an extended binary tree, and the external nodes of the whole tree are exactly the external nodes of the two subtrees — the root is internal, and extending the whole tree extends each subtree in the same way. Therefore

$$X(n) \;=\; X(k) \;+\; X((n-1)-k). \tag{16.3}$$

Apply the hypothesis to both terms of Equation (16.3):

$$X(n) \;=\; (k+1) \;+\; ((n-1)-k+1) \;=\; (k+1)+(n-k) \;=\; n+1.$$

Note that k has canceled: the result does not depend on how the internal nodes are shared out, which is exactly what makes the claim true for *every* shape of tree. By strong induction, $X(n) = n + 1$ for all $n \geq 0$. ■

Check the step on **Figure 16.1**. There $n = 4$ with root B. The left subtree is the single node A, so $k = 1$ and $X(1) = 2$. The right subtree is C with child D, so $(n - 1) - k = 2$ and $X(2) = 3$. Total $2 + 3 = 5 = n + 1$. Correct.

16.5.4 A tempting but careless write-up, and why it loses

The proof above is short, and that makes it easy to write carelessly. Two slips are common, and each one turns a correct argument into one a reader cannot trust.

- **Defining the symbol loosely.** A write-up that opens with *"Let $X(n)$ denote the number of external nodes in a binary tree of n nodes"* has defined the wrong thing: the claim is about n **internal** nodes, and an extended tree with n internal nodes has $2n + 1$ nodes in all. If the rest of the proof silently uses the correct meaning, the argument survives, but the reader must repair it. Define each symbol exactly once, and exactly right.
- **Unbalanced notation.** A displayed line such as $X((n) = X(k) + X((n - 1) - k)$, with an unmatched bracket, forces the reader to guess what was meant. Guesses are how errors propagate. Fix the brackets.

Neither slip affects the mathematics. They are named because the habit of checking your own symbols is what separates a proof that can be trusted from one that must be re-derived by every reader.

16.5.5 Two alternative proofs

A second proof is worth having, both because it may be quicker and because it is an independent check on the first.

Alternative 1 — incremental induction. Induct on n by *building* the tree rather than splitting it. For $n = 0$ the extended tree is one external node, so $X(0) = 1$. Now suppose the claim holds for $n - 1$ and take any extended tree with n internal nodes. Pick an internal node both of whose children are external — one always exists, because following any path downward must eventually reach a node with no internal children. Replace that internal node by a single external node. The

result is an extended binary tree with $n - 1$ internal nodes; we destroyed two external nodes and created one, so we lost exactly one. Hence

$$X(n) - 1 = X(n-1) = n, \qquad \text{so} \qquad X(n) = n + 1.$$

Alternative 2 — counting edges, which is not an induction at all. Every node except the root has exactly one edge to its parent, so a tree with N nodes has $N - 1$ edges. In an extended binary tree every internal node has exactly two children and every external node has none, so the number of edges is also $2n$ — count each edge from above. With $N = n + X(n)$ total nodes,

$$n + X(n) - 1 = 2n \quad \Rightarrow \quad X(n) = n + 1.$$

Two lines. Had the problem said only "prove", this would be a complete answer. Because it says "prove by induction", give the induction and offer this as a check — a sentence such as *"as a check, counting edges from above gives* $2n$ *and from below gives* $n + X(n) - 1$*, which agrees"*.

16.5.6 The experiment

A proof needs no experiment, but building the experiment is how you find out whether you have understood the definitions. **Listing 16.4** constructs random binary trees with a prescribed number of internal nodes and counts both quantities.

Listing 16.4 The experiment behind Problem 16.3: random binary trees of a given internal size, with both counts taken. Note that `externals` returns 1 for a null pointer — that is the definition of an external node, written as code.

```
// Problem 16.3: an extended binary tree with n internal nodes has n+1 external nodes.
// The proof is by induction; this program is the experiment that the proof explains.
#include <iostream>
#include <iomanip>
#include <vector>
#include <random>
using namespace std;

struct Node { Node *l = nullptr, *r = nullptr; };

// Builds a random binary tree with exactly n internal nodes.
Node* build(int n, mt19937& rng) {
    if (n == 0) return nullptr;                 // a null child is an external node
    int k = uniform_int_distribution<int>(0, n - 1)(rng);   // left subtree size
    Node* t = new Node;
    t->l = build(k, rng);
    t->r = build(n - 1 - k, rng);
    return t;
}
int internals(Node* t) { return t ? 1 + internals(t->l) + internals(t->r) : 0; }
int externals(Node* t) { return t ? externals(t->l) + externals(t->r) : 1; }
void freeTree(Node* t) { if (t) { freeTree(t->l); freeTree(t->r); delete t; } }

int main() {
    mt19937 rng(20251214u);                     // a fixed seed: repeatable runs
    cout << "   n   internals   externals   externals - internals\n";
    for (int n : {0, 1, 2, 3, 4, 5, 10, 50, 1000}) {
        Node* t = build(n, rng);
        int in = internals(t), ex = externals(t);
        cout << setw(5) << n << setw(12) << in << setw(12) << ex
             << setw(24) << ex - in << "\n";
        freeTree(t);
    }
    bool ok = true;
    for (int trial = 0; trial < 20000 && ok; ++trial) {
        int n = uniform_int_distribution<int>(0, 60)(rng);
        Node* t = build(n, rng);
        if (externals(t) != internals(t) + 1) ok = false;
        freeTree(t);
    }
    cout << "\n20000 random trees checked: X(n) = n + 1 holds in "
         << (ok ? "every" : "NOT every") << " case\n";
    return 0;
}
```

The output:

```
   n   internals   externals   externals - internals
   0           0           1                       1
   1           1           2                       1
   2           2           3                       1
   3           3           4                       1
   4           4           5                       1
   5           5           6                       1
  10          10          11                       1
  50          50          51                       1
1000        1000        1001                       1

20000 random trees checked: X(n) = n + 1 holds in every case
```

The single most instructive line in the program is `int externals(Node* t) { return t ? ... : 1; }`. The base case of the recursion returns 1 for a null pointer — because a null pointer *is* an external node. That one character is the definition that Section 16.5.1 spent a paragraph giving.

16.5.7 Why the result matters

The problem's own motivation explains it: an external node is a place to store the information you would otherwise lose when a search falls out of the tree. The count $n + 1$ then tells you the cost of storing it — one extra node per internal node, plus one — so an extended tree is less than twice the size of the original.

It also gives, free of charge, one of the most-used facts about binary trees: since every internal node has two children in the extended tree, the total node count is $n + (n + 1) = 2n + 1$, always odd. And it is the combinatorial core of the statement that a binary tree with n internal nodes has exactly $n + 1$ *null links*, which is what makes threaded binary trees possible — every one of those $n + 1$ nulls can be repurposed as a thread to a successor.

16.5.8 The mistakes most often made on this problem

Using **weak** induction — assuming the claim only for $n - 1$ — and then trying to apply it to two subtrees of sizes k and $n - 1 - k$. Unless k happens to be $n - 1$ or 0, neither subtree has $n - 1$ internal nodes, so the hypothesis does not apply and the proof has a hole in exactly the place it needed to be solid.

The fix is one sentence: *"assume $X(n') = n' + 1$ for all $0 \le n' < n$"*. Write it as a line of its own, labeled **Inductive hypothesis**, and the proof cannot develop that hole.

The second is forgetting that the base case is $n = 0$, not $n = 1$. The definition tells you what the empty tree extends to — a single external node — precisely so that you can start at zero. A proof that begins at $n = 1$ is not wrong, but it must then also handle the empty subtree case inside the step, which is more work for no gain.

Check yourself. A *full* binary tree is one in which every node has either zero or two children. Prove, with the same strong-induction skeleton, that a full binary tree with ℓ leaves has exactly $\ell - 1$ nodes with two children.

Answer. Let $I(\ell)$ be the number of two-child nodes. Base, $\ell = 1$: a single node, $I(1) = 0 = 1 - 1$. Step, $\ell \ge 2$: the root has two children, and its subtrees are full with $\ell_1, \ell_2 \ge 1$ leaves, $\ell_1 + \ell_2 = \ell$, both $< \ell$. By strong induction $I(\ell) = 1 + (\ell_1 - 1) + (\ell_2 - 1) = \ell - 1$. This is Problem 16.3 again in disguise: the leaves play the external nodes and the two-child nodes the internal ones.

16.6 Problem 16.4 — A 2-Approximation for the Traveling Salesman

16.6.1 The problem

Problem 16.4. Let $G = (V, E)$ be a complete graph with vertex set $V = \{1,2, \dots, n\}$ and weights $w(i, j) = d_{ij}$, where d_{ij} is the distance between points i and j in an instance of the *Traveling Salesman Problem* (TSP), and assume that the distances satisfy the triangle inequality. Consider the following algorithm. (1) Select a vertex $r \in V$ to serve as the root of a tree. (2) Build a minimum spanning tree T of G rooted at r, using a polynomial-time minimum-spanning-tree algorithm as a subroutine. (3) Construct the list L of vertices visited by a *preorder walk* of T. (4) Return L as an approximate tour of G. Prove:

1. the algorithm is correct, that is, it produces a tour;
2. its running time is polynomial;

3. it has ratio bound $\rho(n) = 2$.

The three parts are not of equal weight. Parts 1 and 2 are a paragraph each; part 3 is the heart of the problem.

16.6.2 What it tests, and the vocabulary you need from nothing

Approximation algorithms are not a chapter of this book — they are one of the six territories named in Section 16.15 as lying beyond it. The problem is nevertheless fair, for a precise reason: **every ingredient of the proof is already in your hands.** It needs a minimum spanning tree (Chapter 11), a preorder walk of a tree (Chapter 10), the triangle inequality (Chapter 13), and the observation that deleting one edge from a tour leaves a spanning tree. What is missing is only vocabulary, and it takes four definitions, each one sentence long. This is what it means to own a toolkit: not remembering answers, but pointing the machinery at something you have not seen.

DEFINITION. A **tour** of a graph $G = (V, E)$ is a cycle that visits every vertex exactly once and returns to its starting point — a *Hamiltonian cycle*. Its **length** is the sum of the weights of its $|V|$ edges. The **Traveling Salesman Problem** asks for a tour of minimum length.

DEFINITION. The weight function w satisfies the **triangle inequality** when, for every three vertices u, v, x,

$$w(u, v) \;\leq\; w(u, x) + w(x, v). \tag{16.4}$$

In words: a detour through x is never a short cut. An instance whose weights satisfy this is called a **metric** instance. Straight-line distance, driving distance and Manhattan distance all satisfy it; a fare table with a discounted connecting flight may not.

Figure 16.2 is the drawing of the inequality from Chapter 13. It was used there for $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$; the picture is the same, and so is the meaning.

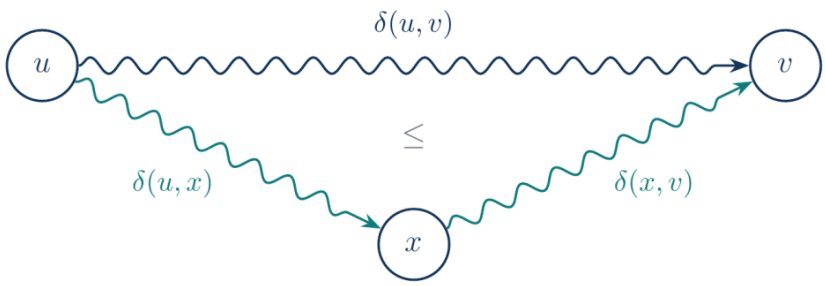


Figure 16.2 The triangle inequality: going from u to v directly is never worse than going from u to x and then from x to v. This single fact is what makes the shortcutting step of Problem 16.4 legal.

DEFINITION. An algorithm for a minimization problem has **ratio bound** $\rho(n)$ if, on every input of size n, the cost C it returns and the optimal cost C^* satisfy $C/C^* \leq \rho(n)$. Such an algorithm is called a $\rho(n)$**-approximation algorithm**. Problem 16.4 asks you to prove $\rho(n) = 2$ — a *constant* ratio, independent of n.

DEFINITION. A **full walk** of a rooted tree T lists every vertex each time it is met: down into a child, and again on the way back up. A **preorder walk** lists every vertex the *first* time it is met. Equivalently, the preorder walk is the full walk with repeated occurrences deleted.

That last equivalence is the sentence that disposes of part 1, so it is worth having in exactly that form.

16.6.3 Part 1 — the algorithm produces a tour

Claim. The list L returned by the algorithm is a tour of G: it contains every vertex of V exactly once.

Proof. T is a spanning tree of G, so its vertex set is V — that is what *spanning* means. A full walk W of T starting at r traverses every edge of T twice, once in each direction, and therefore visits every vertex of T at least once. Since $V(T) = V$, the walk W contains every vertex of G at least once.

The preorder walk L is obtained from W by keeping only the first occurrence of each vertex. Keeping the first occurrence of each vertex of a list that contains every vertex at least once produces a list that contains every vertex **exactly** once. Hence L is a permutation of V.

Finally, G is a **complete** graph, so for any two vertices L_i and L_{i+1} the edge (L_i, L_{i+1}) exists, and so does the closing edge $(L_{|V|}, L_1)$. Therefore the cyclic sequence $L_1, L_2, \ldots, L_{|V|}, L_1$ is a Hamiltonian cycle — a tour. ■

Two remarks worth making explicitly. First, completeness of G is used, and it is used exactly once, here: without it the consecutive vertices of L might not be joined by an edge. Second, the triangle inequality is **not** used in part 1. Whether the weights are metric or not, the algorithm returns a tour; metricity is needed only for part 3. Saying so explicitly shows that you know which hypothesis does which job.

16.6.4 Part 2 — the running time is polynomial

This part is straightforward, but the paragraph should contain numbers, not adjectives.

Let $n = |V|$. Since G is complete, $|E| = \binom{n}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$.

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	$\Theta(n)$

Table 16.4 totals to $\Theta(n^2)$ using Prim with an array, which is the right choice here because a complete graph is as dense as a graph can be, and the $\Theta(n^2)$ array version of Prim beats the $O(E\lg V) = O(n^2 \lg n)$ heap version on dense input. (That is statement (12) of Problem 16.1, arriving as a working decision rather than as an opinion.) Either way, $\Theta(n^2)$ or $O(n^2 \lg n)$ is a polynomial in n, which is what part 2 asks for. ■

It is worth one extra sentence: the *exact* solution of the TSP is not known to be computable in polynomial time — the decision version is NP-complete, and the brute-force search over $(n-1)!/2$ tours is astronomically worse. At $n = 20$ that is about 6×10^{16} tours. The whole point of the algorithm is to trade the exact answer for a polynomial one, and part 2 is what makes the trade worth anything.

16.6.5 Part 3 — the ratio bound is 2

Write it as a chain of four statements, each on its own line, each justified. Notation: T is the minimum spanning tree, W its full walk, L the preorder walk returned by the algorithm, L^* an optimal tour, and $d(\cdot)$ the total weight of a tree, a walk or a tour.

Step 1: $d(T) \leq d(L^*)$ **— the minimum spanning tree is no heavier than the optimal tour.**

Take the optimal tour L^* and delete any one of its edges. A tour is a cycle through all n vertices; deleting one edge from a cycle leaves a path through all n vertices, which is connected and acyclic and touches every vertex — that is, a spanning tree of G. Call it T'. Since all weights are non-negative,

$$d(T') \;\leq\; d(L^*).$$

And since T is a *minimum* spanning tree, it is no heavier than the spanning tree T':

$$d(T) \;\leq\; d(T') \;\leq\; d(L^*).$$

Step 2: $d(W) = 2\,d(T)$ **— the full walk costs exactly twice the tree.**

A full walk enters each subtree once and leaves it once, so it traverses every edge of T exactly twice, once downward and once upward. Summing,

$$d(W) \;=\; \sum_{e \in T} 2\; w(e) \;=\; 2\,d(T).$$

Combining with Step 1,

$$d(W) \;=\; 2\,d(T) \;\leq\; 2\,d(L^*).$$

Step 3: $d(L) \leq d(W)$ **— dropping repeats never makes the trip longer.**

This is where the triangle inequality earns its place, and it is the step most often asserted rather than proved. The preorder list L is obtained from W by deleting repeated vertices. Deleting one repeated vertex means replacing a sub-path $u \to x \to v$ of W by the single edge $u \to v$. By the triangle inequality Equation (16.4),

$$w(u, v) \;\leq\; w(u, x) + w(x, v),$$

so that replacement does not increase the total. L is obtained from W by finitely many such replacements — one per deleted occurrence — and each one is non-increasing, so by induction on the number of deletions

$$d(L) \;\leq\; d(W).$$

Step 4: put them together.

$$d(L) \;\leq\; d(W) \;=\; 2\,d(T) \;\leq\; 2\,d(L^*), \tag{16.5}$$

hence

$$\frac{d(L)}{d(L^*)} \leq 2, \qquad \text{that is} \qquad \rho(n) = 2. \qquad \blacksquare$$

WHERE EACH HYPOTHESIS IS USED. Completeness of G is used in part 1, to guarantee that the closing edges exist. Non-negativity of the weights is used in Step 1, to say that deleting an edge does not increase the total. The triangle inequality is used in Step 3, and nowhere else. Minimality of T is used in Step 1. If you are asked one question about this proof, it will be this one.

16.6.6 What happens without the triangle inequality

Drop it and the bound collapses — not slightly, but completely. Suppose there were a polynomial algorithm with *any* constant ratio ρ for general TSP. Then you could decide whether an arbitrary graph H on n vertices has a Hamiltonian cycle: build a complete graph on the same vertices, give weight 1 to every edge of H and weight $\rho n + 1$ to every edge not in H, and run the approximation. If H has a Hamiltonian cycle, the optimum is n and the algorithm must return at most ρn, so it cannot use any heavy edge. If H has none, every tour uses a heavy edge and costs more than ρn. So the algorithm decides Hamiltonian cycle, which is NP-complete — impossible in polynomial time unless $\mathrm{P} = \mathrm{NP}$.

In one sentence: *"Without the triangle inequality, Step 3 fails, and in fact no constant-ratio polynomial approximation exists unless* $\mathrm{P} = \mathrm{NP}$*."*

16.6.7 A complete worked instance

Abstract proofs are checked on concrete numbers. Take six Profitina Laundry pickup points on the Surabaya street grid and measure in blocks with the Manhattan metric $d(i,j) = |x_i - x_j| + |y_i - y_j|$, which satisfies the triangle inequality:

$$A = (0,0), \quad B = (4,0), \quad C = (8,1), \quad D = (6,5), \quad E = (2,6), \quad F = (0,3).$$

Table 16.5 is the resulting distance matrix. Check one entry by hand: $d(A,D) = |0-6| + |0-5| = 6 + 5 = 11$.

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

Step 2 of the algorithm. A minimum spanning tree rooted at A, by Prim: the cheapest edge out of $\{A\}$ is $A - F$ at 3; then $A - B$ at 4; then $B - C$ at 5; then $F - E$ at 5; then $E - D$ at 5. Total

$$d(T) = 3 + 4 + 5 + 5 + 5 = 22.$$

Step 3. Rooted at A, the tree is $A \to \{B, F\}$, $B \to \{C\}$, $F \to \{E\}$, $E \to \{D\}$. The full walk, taking children in alphabetical order, is

$$W = A,\ B,\ C,\ B,\ A,\ F,\ E,\ D,\ E,\ F,\ A,$$

whose length is $4 + 5 + 5 + 4 + 3 + 5 + 5 + 5 + 5 + 3 = 44$, and indeed $44 = 2 \times 22 = 2\,d(T)$, exactly as Step 2 of the proof predicts.

Deleting repeats gives the preorder list

$$L = A,\ B,\ C,\ F,\ E,\ D,$$

and the tour $A \to B \to C \to F \to E \to D \to A$ costs

$$4 + 5 + 10 + 5 + 5 + 11 = 40.$$

The optimum. With six vertices there are $5!/2 = 60$ distinct tours, so brute force is feasible. The best is $A \to B \to C \to D \to E \to F \to A$ at

$$4 + 5 + 6 + 5 + 5 + 3 = 28.$$

All four inequalities of Equation (16.5)**, checked:**

$$d(T) = 22 \ \le\ d(L^*) = 28 \quad \checkmark$$

$$d(W) = 44 \ =\ 2\,d(T) = 44 \quad \checkmark$$

$$d(L) = 40 \ \le\ d(W) = 44 \quad \checkmark$$

$$\frac{d(L)}{d(L^*)} = \frac{40}{28} = 1.4286 \ \le\ 2 \quad \checkmark$$

16.6.8 The program

Listing 16.5 does all of the above and prints every intermediate quantity, so that none of the numbers in Section 16.6.7 has to be taken on trust.

Listing 16.5 The complete approximation algorithm of Problem 16.4, with the triangle inequality verified, the minimum spanning tree built by Prim, the full and preorder walks printed, and the optimum found by brute force for comparison.

```
// Problem 16.4: the MST-plus-preorder-walk 2-approximation for metric TSP,
// run on six Profitina Laundry pickup points with the Manhattan metric,
// and compared against the optimal tour found by brute force.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
#include <numeric>
#include <climits>
using namespace std;

const int N = 6;
const string name[N]  = {"A","B","C","D","E","F"};
const int  x[N] = {0, 4, 8, 6, 2, 0};
const int  y[N] = {0, 0, 1, 5, 6, 3};

int d[N][N];

vector<int> child[N];                        // the MST, rooted
vector<int> preorderList, fullWalk;

void preorder(int u) {
    preorderList.push_back(u);
    fullWalk.push_back(u);
    for (int v : child[u]) { preorder(v); fullWalk.push_back(u); }
}

int tourCost(const vector<int>& t) {
    int s = 0;
    for (size_t i = 0; i < t.size(); ++i) s += d[t[i]][t[(i + 1) % t.size()]];
    return s;
}
int walkCost(const vector<int>& w) {
    int s = 0;
    for (size_t i = 0; i + 1 < w.size(); ++i) s += d[w[i]][w[i + 1]];
    return s;
}

int main() {
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            d[i][j] = abs(x[i] - x[j]) + abs(y[i] - y[j]);   // Manhattan: a metric
```

Listing 16.5 (Continued) *The complete approximation algorithm of Problem 16.4, with the triangle inequality verified, the minimum spanning tree built by Prim, the full and preorder walks printed, and the optimum found by brute force for comparison.*

```
    cout << "distance matrix (Manhattan)\n    ";
    for (int j = 0; j < N; ++j) cout << setw(3) << name[j];
    cout << "\n";
    for (int i = 0; i < N; ++i) {
        cout << setw(4) << name[i];
        for (int j = 0; j < N; ++j) cout << setw(3) << d[i][j];
        cout << "\n";
    }

    bool tri = true;                        // the hypothesis the proof needs
    for (int i = 0; i < N; ++i) for (int j = 0; j < N; ++j) for (int k = 0; k < N; ++k)
        if (d[i][j] > d[i][k] + d[k][j]) tri = false;
    cout << "\ntriangle inequality holds: " << (tri ? "yes" : "no") << "\n";

    // Step 2: a minimum spanning tree rooted at r = A, by Prim.
    const int r = 0;
    vector<int> key(N, INT_MAX), par(N, -1);
    vector<bool> in(N, false);
    key[r] = 0;
    int mstWeight = 0;
    for (int it = 0; it < N; ++it) {
        int u = -1;
        for (int v = 0; v < N; ++v) if (!in[v] && (u == -1 || key[v] < key[u])) u = v;
        in[u] = true;
        mstWeight += key[u];
        if (par[u] != -1) cout << "MST edge " << name[par[u]] << "-" << name[u]
                               << "  w = " << key[u] << "\n";
        for (int v = 0; v < N; ++v)
            if (!in[v] && d[u][v] < key[v]) { key[v] = d[u][v]; par[v] = u; }
    }
    cout << "MST weight d(T) = " << mstWeight << "\n";

    for (int v = 0; v < N; ++v) if (par[v] != -1) child[par[v]].push_back(v);
    for (int v = 0; v < N; ++v) sort(child[v].begin(), child[v].end());

    // Steps 3 and 4: the full walk, then the preorder walk, then the tour.
    preorder(r);
    cout << "\nfull walk W    :";
    for (int v : fullWalk) cout << " " << name[v];
    cout << "\n  d(W) = " << walkCost(fullWalk) << "  (= 2 d(T) = " << 2 * mstWeight <<
")\n";
    cout << "preorder list L:";
    for (int v : preorderList) cout << " " << name[v];
    cout << "\n  d(L) = " << tourCost(preorderList) << "\n";
```

Listing 16.5 (Continued) *The complete approximation algorithm of Problem 16.4, with the triangle inequality verified, the minimum spanning tree built by Prim, the full and preorder walks printed, and the optimum found by brute force for comparison.*

```
    // The optimum, by brute force over all (N-1)! rotations fixed at A.
    vector<int> perm(N);
    iota(perm.begin(), perm.end(), 0);
    int best = INT_MAX; vector<int> bestTour;
    do {
        if (perm[0] != 0) continue;
        int c = tourCost(perm);
        if (c < best) { best = c; bestTour = perm; }
    } while (next_permutation(perm.begin(), perm.end()));

    cout << "\noptimal tour L*:";
    for (int v : bestTour) cout << " " << name[v];
    cout << "\n  d(L*) = " << best << "\n";
    cout << fixed << setprecision(4)
         << "\nratio d(L)/d(L*) = " << (double)tourCost(preorderList) / best
         << "   (the theorem guarantees at most 2)\n"
         << "check d(T) <= d(L*):    " << mstWeight << " <= " << best << "\n"
         << "check d(L) <= 2 d(L*):  " << tourCost(preorderList) << " <= " << 2 * best <<
"\n";
    return 0;
}
```

Compiled and run, it prints:

```
distance matrix (Manhattan)
      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

triangle inequality holds: yes
MST edge A-F  w = 3
MST edge A-B  w = 4
MST edge B-C  w = 5
MST edge F-E  w = 5
MST edge E-D  w = 5
MST weight d(T) = 22

full walk W    : A B C B A F E D E F A
  d(W) = 44  (= 2 d(T) = 44)
preorder list L: A B C F E D
  d(L) = 40

optimal tour L*: A B C D E F
  d(L*) = 28

ratio d(L)/d(L*) = 1.4286   (the theorem guarantees at most 2)
check d(T) <= d(L*):    22 <= 28
check d(L) <= 2 d(L*):  40 <= 56
```

16.6.9 How tight is 2?

The bound is *proved*, so the ratio can never exceed 2. It is also nearly *tight*: there are metric instances on which the ratio approaches 2 as n grows. Our six points gave 1.4286; the six points of Exercise 4 in Section 16.19 give 1.5556; families are known that approach $2 - \varepsilon$ for every $\varepsilon > 0$. So the 2 is not an artifact of a lazy proof. The honest summary is: *"the bound is 2, it is asymptotically tight for this algorithm, and on typical instances the observed ratio is well below it."*

If you want a better guarantee, Christofides' algorithm [2] replaces the doubling of the tree by a minimum-weight perfect matching on the odd-degree vertices of T, and achieves $\rho = 3/2$ in polynomial time. It was the best known constant for metric TSP from 1976 until 2021, when Karlin, Klein and Oveis Gharan [3] beat it by a margin smaller than 10^{-36}; later work by Gurvits, Klein and Leake [4] improved that margin only slightly, so as of September 2026 the best guarantee for metric TSP is still 3/2 for every practical purpose. One sentence naming it places the algorithm in its field.

16.6.10 The mistakes most often made on this problem

The first is proving $d(L) \leq d(W)$ by saying "removing vertices makes the walk shorter". It does not, in general — removing a vertex from a walk *replaces two edges with one*, and without the triangle inequality the one replacement edge can be arbitrarily heavier than the two it replaces. The statement is true here **because the instance is metric**, and the proof must say so, in the place where it is used.

The second is proving $d(T) \leq d(L^*)$ by asserting "a tree is cheaper than a tour". That is not a theorem; it is the conclusion. The argument is: *delete an edge from the optimal tour to get a spanning tree, whose weight is at most the tour's because weights are non-negative, and T is minimum among spanning trees, so $d(T)$ is at most that.* Three clauses. Write all three.

Check yourself. In the instance of Section 16.6.7, suppose the preorder walk took the children of A in the order F before B. What tour results, what does it cost, and does Equation (16.5) still hold?

Answer. The full walk becomes $A, F, E, D, E, F, A, B, C, B, A$, still of length 44; the preorder list is A, F, E, D, B, C, and the tour costs $3 + 5 + 5 + 7 + 5 + 9 = 34$. Then $34 \leq 44 = 2 \cdot 22 \leq 56$, so the chain holds — as it must, since the proof never used the order of the children. The ratio $34/28 = 1.2143$ shows that the choice of child order, which the algorithm leaves free, can matter a great deal in practice.

16.7 Problem 16.5 — Bellman–Ford against Dijkstra

16.7.1 The problem

Problem 16.5. Bellman–Ford and Dijkstra are two algorithms that find shortest paths from one vertex to all other vertices of a weighted graph.

1. Describe their differences and their respective real-life applications.
2. Write pseudocode for each algorithm and show where they differ.
3. Give a short illustration of how each algorithm works.

16.7.2 What it tests

Whether you understand that the two algorithms are **the same idea with two different orderings**. Both initialize $d[s] = 0$ and $d[v] = \infty$, and both improve estimates by relaxation, Equation (16.1). Everything after that is a choice of **which edge to relax next, and how many times**. Dijkstra relaxes the edges out of the closest unfinished vertex, once each. Bellman–Ford relaxes every edge, $|V| - 1$ times. An answer that says "one uses a priority queue and the other uses loops" has described the code; an answer that says "they differ only in the order and

multiplicity of relaxations, and that difference is exactly what negative weights break" has described the algorithms.

16.7.3 Part 1 — the differences, and the applications

The one difference that no answer may omit is that Bellman–Ford handles edges of negative weight and Dijkstra does not. **Table 16.6** is what a full answer looks like.

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	$\Theta(VE)$
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)

Real-life applications, with the reason attached. An application without its reason is a list of names; here are the reasons.

- **Bellman–Ford: distance-vector routing.** The Routing Information Protocol (RIP), and the distance-vector family generally, is Bellman–Ford run in a distributed way: each router keeps a vector of estimated distances and periodically sends it to its neighbors, who relax their own estimates against it. Bellman–Ford is the right algorithm here not because link costs are negative — they are not — but because of the *distributed* row in **Table 16.6**: relaxing an edge needs only the two routers at its ends, whereas Dijkstra's `Extract-Min` needs a global view that no single router has.
- **Bellman–Ford: currency arbitrage.** Take exchange rates r_{ij} and set $w(i,j) = -\ln r_{ij}$. A sequence of trades that multiplies your money by more than one becomes a cycle of negative total weight, and Bellman–Ford's negative-cycle test finds it. This is the standard example of *genuinely* negative weights, and it answers the obvious objection that distances are never negative.
- **Dijkstra: map and navigation applications.** Road distances and travel times are non-negative, graphs are sparse, and answers are wanted in milliseconds. Dijkstra with a binary heap — or its goal-directed relatives A^* and contraction hierarchies — is what every navigation product runs.
- **Dijkstra: link-state routing.** OSPF floods the whole topology to every router, and then each router runs Dijkstra locally on the complete map. Note the trade: OSPF pays the cost of

distributing the topology in order to buy Dijkstra's speed, which is exactly the opposite decision from RIP's.

16.7.4 Part 2 — the two pieces of pseudocode

Figure 16.3 and **Figure 16.4** are the pseudocode of the two algorithms as Chapter 13 gave them. Reproducing them is not the point of part 2; *annotating the differences* is.

```
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.

```
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.

Put them side by side and four differences fall out. Present them as a numbered list — it is the difference between a description and an analysis.

1. **Lines 1–3 are identical.** Both set $d[v] \leftarrow \infty$ for every v and $d[s] \leftarrow 0$. The initialization is not where they differ, and saying so first shows that you know what is shared.
2. **Dijkstra lines 4–7 versus Bellman–Ford line 5.** Dijkstra maintains a set S of finished vertices and a priority queue Q, and chooses the next vertex by `Extract-Min`. Bellman–Ford chooses nothing: it loops $|V| - 1$ times over the whole edge list. Dijkstra's choice is the greedy choice; Bellman–Ford's loop is the dynamic program.
3. **Dijkstra relaxes each edge once; Bellman–Ford relaxes each edge $|V| - 1$ times.** Dijkstra's line 9 iterates over $Adj[u]$ for each u exactly once over the whole run, so every edge is relaxed exactly once. Bellman–Ford's line 6 is inside the line 5 loop, so every edge is relaxed $|V| - 1$ times. That single structural fact is the whole running-time difference: $O(E\lg V)$ against $\Theta(VE)$.
4. **Bellman–Ford has lines 8–10; Dijkstra has nothing like them.** The closing sweep asks whether any edge can *still* be relaxed after $|V| - 1$ passes. If one can, a shortest path would need $\geq |V|$ edges, which means it repeats a vertex, which means there is a cycle of negative weight; the algorithm returns `false`. Dijkstra cannot do this, because it has already thrown the information away — a vertex it has finalized is never looked at again.

Why $|V| - 1$ passes suffice. This is the sentence that converts a description into a proof, and it is cheap. A shortest path with no negative cycle never repeats a vertex, so it has at most $|V| - 1$ edges. After pass i, $d[v]$ is correct for every v whose shortest path uses at most i edges — by induction, since pass i relaxes every edge and so in particular the last edge of that path. Hence after $|V| - 1$ passes every $d[v]$ is correct.

16.7.5 Part 3 — the illustrations

Part 3 asks for a *short* illustration: two pictures and two tables of numbers is exactly right, and anything more is gold-plating.

Dijkstra. **Figure 16.5** is the Chapter 13 graph at initialization, with $d[s] = 0$ and every other estimate ∞.

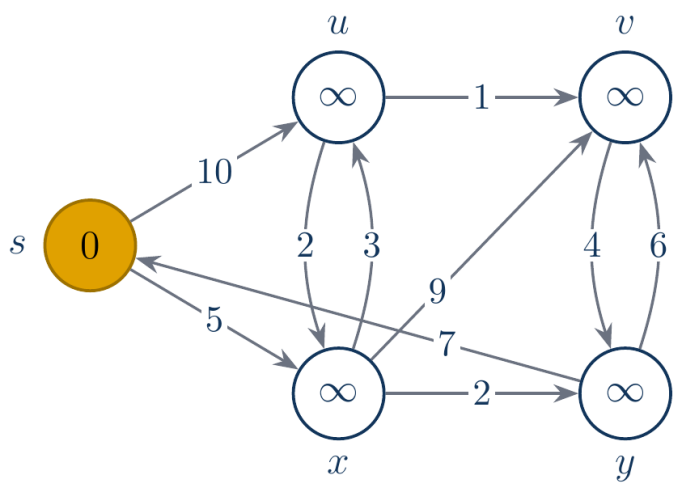


Figure 16.5 Dijkstra's example at initialization: $d[s] = 0$, and $d[u] = d[v] = d[x] = d[y] = \infty$. Edge weights $s \to u = 10$, $s \to x = 5$, $u \to v = 1$, $u \to x = 2$, $x \to u = 3$, $x \to v = 9$, $x \to y = 2$, $y \to v = 6$, $y \to s = 7$, $v \to y = 4$.

Now the trace, one line per `Extract-Min`, with the arithmetic:

- **Extract** s ($d = 0$). Relax $s \to u$: $0 + 10 = 10 < \infty$, so $d[u] = 10$. Relax $s \to x$: $0 + 5 = 5 < \infty$, so $d[x] = 5$.
- **Extract** x ($d = 5$, the smallest). Relax $x \to u$: $5 + 3 = 8 < 10$, so $d[u] = 8$. Relax $x \to v$: $5 + 9 = 14 < \infty$, so $d[v] = 14$. Relax $x \to y$: $5 + 2 = 7 < \infty$, so $d[y] = 7$.
- **Extract** y ($d = 7$). Relax $y \to v$: $7 + 6 = 13 < 14$, so $d[v] = 13$. Relax $y \to s$: $7 + 7 = 14 \not< 0$, no change.
- **Extract** u ($d = 8$). Relax $u \to v$: $8 + 1 = 9 < 13$, so $d[v] = 9$. Relax $u \to x$: $8 + 2 = 10 \not< 5$, no change.
- **Extract** v ($d = 9$). Relax $v \to y$: $9 + 4 = 13 \not< 7$, no change. Queue empty.

Figure 16.6 is the state on termination, with every vertex navy — finalized — and the shortest-path tree in bold.

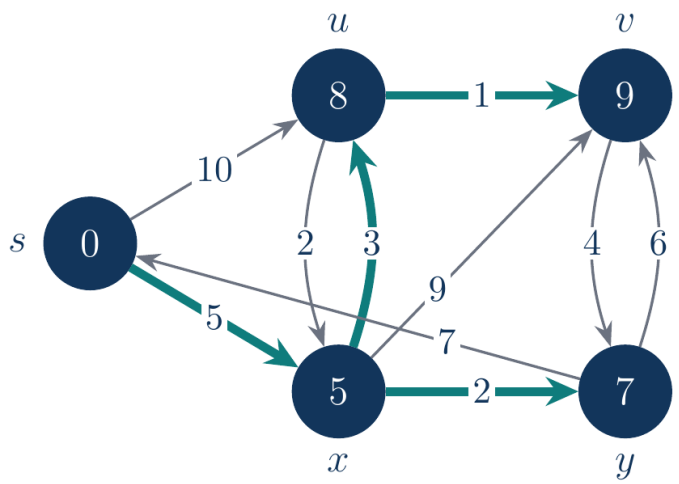


Figure 16.6 Dijkstra's example on termination: $d[s] = 0$, $d[x] = 5$, $d[y] = 7$, $d[u] = 8$, $d[v] = 9$. The bold edges are the shortest-path tree. Compare the numbers with the hand trace in Section 16.7.5.

Bellman–Ford. Figure 16.7 is the negative-weight graph of Chapter 13 at initialization.

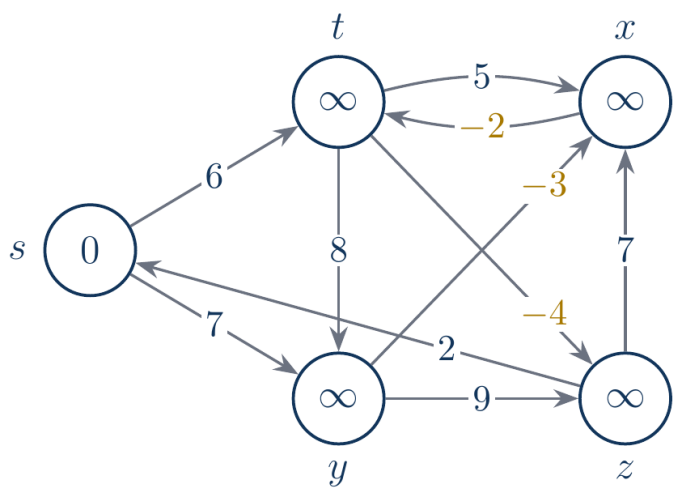


Figure 16.7 Bellman–Ford's example at initialization. Weights: $s \to t = 6$, $s \to y = 7$, $t \to x = 5$, $t \to y = 8$, $t \to z = -4$, $x \to t = -2$, $y \to x = -3$, $y \to z = 9$, $z \to s = 2$, $z \to x = 7$. Three of the ten weights are negative, so Dijkstra is not applicable.

With the edges relaxed in the listed order, the passes go as in **Table 16.7**.

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

Check pass 1 by hand. Starting from $d[s] = 0$ with everything else infinite, the relaxations in order are: $s \to t$ gives $d[t] = 6$; $s \to y$ gives $d[y] = 7$; $t \to x$ gives $d[x] = 6 + 5 = 11$; $t \to y$ gives $6 + 8 = 14 \not< 7$, no change; $t \to z$ gives $d[z] = 6 - 4 = 2$; $x \to t$ gives $11 - 2 = 9 \not< 6$; $y \to x$ gives $7 - 3 = 4 < 11$, so $d[x] = 4$; $y \to z$ gives $7 + 9 = 16 \not< 2$; $z \to s$ gives $2 + 2 = 4 \not< 0$; $z \to x$ gives $2 + 7 = 9 \not< 4$. End of pass 1: (0,6,4,7,2) — as the table says.

Figure 16.8 is the state on termination.

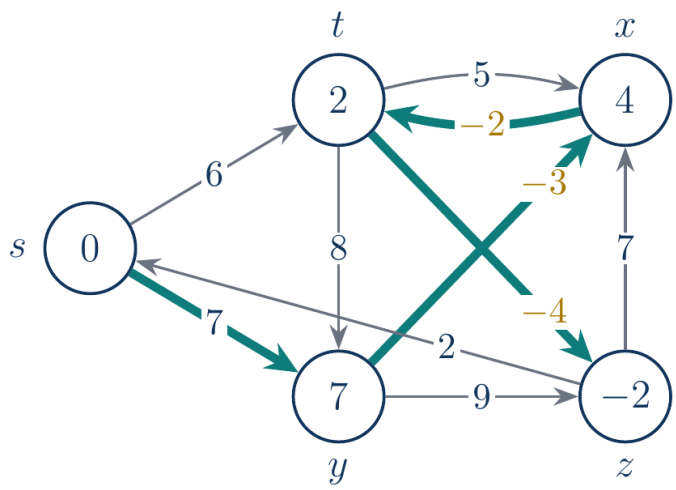


Figure 16.8 Bellman–Ford on termination: $d[s] = 0$, $d[t] = 2$, $d[x] = 4$, $d[y] = 7$, $d[z] = -2$. The bold edges are the shortest-path tree. The path to t is $s \to y \to x \to t$, costing $7 - 3 - 2 = 2$ — three edges, where the direct edge $s \to t$ costs 6.

16.7.6 The program

Listing 16.6 runs both algorithms on the two graphs of Chapter 13 and prints the estimate vector after every step, so that **Table 16.7** and the Dijkstra trace can be checked mechanically.

Listing 16.6 Both algorithms of Problem 16.5 on the two graphs of Chapter 13, printing the estimate vector after each extraction and after each pass.

```
// Problem 16.5: Dijkstra and Bellman-Ford, run on the two example graphs of
// Chapter 13, with the estimate d[v] printed after every step.
#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; };

void show(const string& tag, const vector<string>& nm, const vector<int>& d) {
    cout << setw(14) << tag << " :";
    for (size_t i = 0; i < d.size(); ++i) {
        cout << "  " << nm[i] << "=";
        if (d[i] >= INF) cout << setw(3) << "inf"; else cout << setw(3) << d[i];
    }
    cout << "\n";
}

void dijkstra(const vector<string>& nm, const vector<Edge>& E, int s) {
    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); d[s] = 0;
    vector<bool> done(n, false);
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    Q.push({0, s});
    show("initialize", nm, d);
    while (!Q.empty()) {
        auto [du, u] = Q.top(); Q.pop();
        if (done[u]) continue;                // a stale entry: skip it
        done[u] = true;
        for (auto [v, w] : adj[u])
            if (d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
        show("extract " + nm[u], nm, d);
    }
}
```

Listing 16.6 (Continued) *Both algorithms of Problem 16.5 on the two graphs of Chapter 13, printing the estimate vector after each extraction and after each pass.*

```
bool bellmanFord(const vector<string>& nm, const vector<Edge>& E, int s) {
    int n = (int)nm.size();
    vector<int> d(n, INF); d[s] = 0;
    show("initialize", nm, d);
    for (int pass = 1; pass <= n - 1; ++pass) {
        for (auto& e : E)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) d[e.v] = d[e.u] + e.w;
        show("pass " + to_string(pass), nm, d);
    }
    for (auto& e : E)
        if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) return false;   // negative cycle
    return true;
}

int main() {
    cout << "=== Dijkstra on the Chapter 13 graph (s,u,v,x,y) ===\n";
    vector<string> nmD = {"s","u","v","x","y"};
    vector<Edge> ED = {{0,1,10},{0,3,5},{1,2,1},{1,3,2},{3,1,3},
                       {3,2,9},{3,4,2},{4,2,6},{4,0,7},{2,4,4}};
    dijkstra(nmD, ED, 0);

    cout << "\n=== Bellman-Ford on the Chapter 13 graph (s,t,x,y,z) ===\n";
    vector<string> nmB = {"s","t","x","y","z"};
    vector<Edge> EB = {{0,1,6},{0,3,7},{1,2,5},{1,3,8},{1,4,-4},
                       {2,1,-2},{3,2,-3},{3,4,9},{4,0,2},{4,2,7}};
    bool ok = bellmanFord(nmB, EB, 0);
    cout << "returns " << (ok ? "true" : "false") << " (no negative cycle)\n";
    return 0;
}
```

Compiled with `g++ -std=c++17 -O2 -Wall` and run:

```
=== Dijkstra on the Chapter 13 graph (s,u,v,x,y) ===
    initialize :  s=  0  u=inf  v=inf  x=inf  y=inf
     extract s :  s=  0  u= 10  v=inf  x=  5  y=inf
     extract x :  s=  0  u=  8  v= 14  x=  5  y=  7
     extract y :  s=  0  u=  8  v= 13  x=  5  y=  7
     extract u :  s=  0  u=  8  v=  9  x=  5  y=  7
     extract v :  s=  0  u=  8  v=  9  x=  5  y=  7

=== Bellman-Ford on the Chapter 13 graph (s,t,x,y,z) ===
    initialize :  s=  0  t=inf  x=inf  y=inf  z=inf
        pass 1 :  s=  0  t=  6  x=  4  y=  7  z=  2
        pass 2 :  s=  0  t=  2  x=  4  y=  7  z=  2
        pass 3 :  s=  0  t=  2  x=  4  y=  7  z= -2
        pass 4 :  s=  0  t=  2  x=  4  y=  7  z= -2
returns true (no negative cycle)
```

Every value matches **Figure 16.6**, **Figure 16.8** and **Table 16.7**.

16.7.7 Showing *why* Dijkstra fails, not just *that* it does

The difference is more convincing if you can exhibit it. **Listing 16.7** does two things: it shows a four-vertex graph on which Dijkstra returns a wrong distance, and it shows Bellman–Ford refusing a graph with a negative cycle.

Listing 16.7 Dijkstra's failure on a negative edge, and Bellman–Ford's negative-cycle detection, on the two smallest graphs that show each.

```
// Why Dijkstra needs non-negative weights, and what Bellman-Ford does instead.
// Part 1: a four-vertex graph with one negative edge on which Dijkstra reports a
//         distance that is simply wrong.
// Part 2: a graph with a negative-weight cycle, which Bellman-Ford detects.
#include <iostream>
#include <string>
#include <vector>
#include <climits>
using namespace std;

const int INF = INT_MAX / 4;
struct Edge { int u, v, w; };

// Dijkstra, with the value FROZEN at the moment a vertex leaves the queue --
// which is exactly what the algorithm promises and what a caller would read.
vector<int> dijkstra(int n, const vector<Edge>& E, int s) {
    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), frozen(n, INF);
    vector<bool> done(n, false);
    d[s] = 0;
    for (int it = 0; it < n; ++it) {
        int u = -1;
        for (int v = 0; v < n; ++v) if (!done[v] && (u == -1 || d[v] < d[u])) u = v;
        if (u == -1 || d[u] >= INF) break;
        done[u] = true;
        frozen[u] = d[u];                       // committed: never revisited
        for (auto [v, w] : adj[u])
            if (!done[v] && d[u] + w < d[v]) d[v] = d[u] + w;
    }
    return frozen;
}

bool bellmanFord(int n, const vector<Edge>& E, int s, vector<int>& d) {
    d.assign(n, INF); d[s] = 0;
    for (int i = 1; i <= n - 1; ++i)
        for (auto& e : E)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) d[e.v] = d[e.u] + e.w;
    for (auto& e : E)
        if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) return false;
    return true;
}

int main() {
    const string nm[4] = {"s", "a", "b", "c"};
```

Listing 16.7 (Continued) *Dijkstra's failure on a negative edge, and Bellman–Ford's negative-cycle detection, on the two smallest graphs that show each.*

```
    // s->a 2, s->b 3, b->a -2, a->c 1.  True: s->b->a->c costs 3-2+1 = 2.
    vector<Edge> G1 = {{0,1,2}, {0,2,3}, {2,1,-2}, {1,3,1}};
    vector<int> dj = dijkstra(4, G1, 0), bf;
    bellmanFord(4, G1, 0, bf);
    cout << "graph 1:  s->a = 2,  s->b = 3,  b->a = -2,  a->c = 1\n";
    cout << "  vertex   Dijkstra   Bellman-Ford\n";
    for (int i = 0; i < 4; ++i)
        cout << "     " << nm[i] << "          " << dj[i] << "             " << bf[i] <<
"\n";
    cout << "  the true shortest path to c is s->b->a->c = 3 + (-2) + 1 = 2;\n"
         << "  Dijkstra froze d[a] = 2 before the edge b->a could improve it.\n";

    // s->a 2, a->b 1, b->a -4: the cycle a->b->a has weight -3.
    vector<Edge> G2 = {{0,1,2}, {1,2,1}, {2,1,-4}};
    vector<int> d3;
    bool ok = bellmanFord(3, G2, 0, d3);
    cout << "\ngraph 2:  s->a = 2,  a->b = 1,  b->a = -4   (cycle weight -3)\n"
         << "  Bellman-Ford returns " << (ok ? "true" : "false")
         << " -- a negative-weight cycle is reachable from s\n";
    return 0;
}
```

Its output:

```
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
```

Follow graph 1 by hand and the mechanism is plain. Dijkstra extracts s, sets $d[a] = 2$ and $d[b] = 3$, then extracts a — because $2 < 3$ — declares $d[a] = 2$ final, and relaxes $a \to c$ to give $d[c] = 3$. Only afterward does it extract b and discover that $s \to b \to a$ costs $3 - 2 = 1$. But a is finished; its outgoing edge is never relaxed again, so $d[c]$ stays at 3 when the true answer is 2. **Dijkstra's greedy choice is only safe when extending a path cannot make it shorter — that is, when weights are non-negative.**

16.7.8 The mistakes most often made on this problem

The first is writing that "Bellman–Ford is slower, so use Dijkstra whenever you can" and stopping. It is a true sentence, but it names one row of **Table 16.6** and no others. The rows that separate a good answer from an adequate one are *negative cycles are detected* and *each node needs only its neighbors* — the second is why routers use an algorithm that is asymptotically worse, which is statement (12) of Problem 16.1 again, in a real system.

The second concerns part 3: drawing the graph but not the numbers. An illustration of a shortest-path algorithm that does not show $d[v]$ changing is an illustration of a graph, not of an algorithm. Put the estimates in the nodes, as **Figure 16.5** and **Figure 16.8** do, and show at least two intermediate states.

Check yourself. In graph 1 of **Listing 16.7**, change the weight of $b \to a$ from -2 to $+2$. What do Dijkstra and Bellman–Ford now report, and why do they agree?

Answer. Both report $d = (0,2,3,3)$ for (s, a, b, c). With all weights non-negative, the greedy choice is safe: when a is extracted at 2, no path through b can beat it, since any such path already costs at least $d[b] = 3 > 2$ before its last edge — which now weighs $+2$, not -2.

16.8 Worked Problems on Single-Source Shortest Paths

Problem 16.5 compared the two great single-source algorithms in words. The three problems of this section make you *run* them, by hand, on graphs you have not seen before, with every estimate and every predecessor written down after every step. They exercise Chapter 13 in the order it was taught: Dijkstra, then Bellman–Ford with its negative-cycle test, then the DAG algorithm.

16.8.1 Problem 16.6 — Dijkstra, with the whole table after every iteration

Problem 16.6. Run Dijkstra's algorithm from s on the directed graph with the ten edges

$$s \to a\ (4),\quad s \to b\ (1),\quad b \to a\ (2),\quad a \to c\ (5),\quad b \to c\ (8),\quad b \to d\ (10),\quad c \to d\ (2),\quad c \to e\ (6),\quad d \to e\ (1),\quad a \to d\ (9).$$

After every `Extract-Min`, give the estimate $d[v]$ and the predecessor $\pi[v]$ of every vertex. Then give the shortest-path tree, the shortest path to every vertex, and a count of the relaxations that succeeded.

What it tests. The mechanics of Dijkstra's algorithm (Section 13.7): the choice of the vertex with the smallest estimate among those not yet finished, the relaxation of its out-edges by Equation (16.1), and the bookkeeping of the predecessor array (Section 13.10) that turns a table of numbers into actual routes.

Solution. All weights are non-negative, so Dijkstra is applicable. Write S for the set of finished vertices. Initialization sets $d[s] = 0$, every other $d[v] = \infty$, every $\pi[v] = \text{NIL}$, and $S = \emptyset$.

Iteration 1. The unfinished estimates are s: 0, all others ∞; extract s. $S = \{s\}$. Relax its out-edges:

- $s \to a$: $d[s] + 4 = 0 + 4 = 4 < \infty$, so $d[a] \leftarrow 4, \pi[a] \leftarrow s$.
- $s \to b$: $0 + 1 = 1 < \infty$, so $d[b] \leftarrow 1, \pi[b] \leftarrow s$.

Iteration 2. Unfinished: a: 4, b: 1, c, d, e: ∞. The smallest is b at 1; extract it. $S = \{s, b\}$.

- $b \to a$: $1 + 2 = 3 < 4$, so $d[a] \leftarrow 3$, $\pi[a] \leftarrow b$. The first improvement of an estimate that was already finite: the direct edge $s \to a$ of weight 4 has lost to the two-edge route $s \to b \to a$ of weight 3.
- $b \to c$: $1 + 8 = 9 < \infty$, so $d[c] \leftarrow 9, \pi[c] \leftarrow b$.
- $b \to d$: $1 + 10 = 11 < \infty$, so $d[d] \leftarrow 11, \pi[d] \leftarrow b$.

Iteration 3. Unfinished: a: 3, c: 9, d: 11, e: ∞. Extract a at 3. $S = \{s, b, a\}$.

- $a \to c$: $3 + 5 = 8 < 9$, so $d[c] \leftarrow 8, \pi[c] \leftarrow a$.
- $a \to d$: $3 + 9 = 12 \not< 11$. No change. This is the one relaxation in the whole run that fails.

Iteration 4. Unfinished: c: 8, d: 11, e: ∞. Extract c at 8. $S = \{s, b, a, c\}$.

- $c \to d$: $8 + 2 = 10 < 11$, so $d[d] \leftarrow 10, \pi[d] \leftarrow c$.
- $c \to e$: $8 + 6 = 14 < \infty$, so $d[e] \leftarrow 14, \pi[e] \leftarrow c$.

Iteration 5. Unfinished: d: 10, e: 14. Extract d at 10. $S = \{s, b, a, c, d\}$.

- $d \to e$: $10 + 1 = 11 < 14$, so $d[e] \leftarrow 11, \pi[e] \leftarrow d$.

Iteration 6. Extract e at 11. It has no out-edges. The queue is empty and the algorithm stops.

Table 16.8 collects the six iterations. Each cell is written $d[v]/\pi[v]$, with $-$ for NIL, and a cell is in bold when its vertex is extracted in that row — from then on, by the correctness theorem of Section 13.7.8, its value never changes.

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**

Answer. $\delta(s,\cdot) = (s{:}\,0,\ a{:}\,3,\ b{:}\,1,\ c{:}\,8,\ d{:}\,10,\ e{:}\,11)$. Following π backward from each vertex gives the routes:

$$e \leftarrow d \leftarrow c \leftarrow a \leftarrow b \leftarrow s, \qquad \text{so the path to } e \text{ is } s \to b \to a \to c \to d \to e,\ \ 1+2+5+2+1 = 11.$$

Here the shortest-path tree happens to be a single path, $s \to b \to a \to c \to d \to e$; every vertex's route is a prefix of it. Of the ten edges, every one was relaxed exactly once, and nine relaxations succeeded; four of those nine (at a, c, d and e) lowered an estimate that was already finite, which in a heap implementation is a `Decrease-Key`.

Checking the answer independently. Take d, the vertex with the most competing routes, and list them all: $s \to b \to d$ costs $1 + 10 = 11$; $s \to a \to d$ costs $4 + 9 = 13$; $s \to b \to a \to d$ costs $1 + 2 + 9 = 12$; $s \to b \to c \to d$ costs $1 + 8 + 2 = 11$; $s \to a \to c \to d$ costs $4 + 5 + 2 = 11$; $s \to b \to a \to c \to d$ costs $1 + 2 + 5 + 2 = 10$. The minimum is 10, as the table says. Notice that three different routes tie at 11; Dijkstra held one of them, $s \to b \to d$, for two iterations before the route through a and c replaced it.

Program. **Listing 16.8** prints the same table.

Listing 16.8 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.

```
// 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 16.8 (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;
}
```

It prints:

```
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, the output is **Table 16.8**.

Check yourself. Change the weight of $s \to b$ from 1 to 5 and run the algorithm again. Give the extraction order and the resulting d/π values.

Answer. The extraction order becomes s, a, b, c, d, e, and the values on termination are $a: 4/s$, $b: 5/s$, $c: 9/a$, $d: 11/c$, $e: 12/d$. When b is extracted at 5, all three of its relaxations fail — $5 + 2 = 7 \not< 4$, $5 + 8 = 13 \not< 9$, $5 + 10 = 15 \not< 13$ — so b ends as a leaf of the shortest-path tree, which is now $s \to a \to c \to d \to e$ together with the separate branch $s \to b$.

16.8.2 Problem 16.7 — Bellman–Ford, and a negative cycle found and exhibited

Problem 16.7. A directed graph on the vertices s, a, b, c, d has the seven edges below, and Bellman–Ford relaxes them in exactly this order in every pass:

$$(b, c, -2), \quad (a, b, 4), \quad (s, a, 3), \quad (s, b, 6), \quad (c, d, 3), \quad (a, c, 5), \quad (d, b, 1).$$

(a) Run Bellman–Ford from s, giving d/π after each of the $|V| - 1 = 4$ passes, and the result of the closing check. (b) Now change the weight of $d \to b$ from 1 to -2 and run it again. Show that the closing check fails, and use the predecessor array to exhibit a negative-weight cycle.

What it tests. Bellman–Ford exactly as Section 13.8 gave it, including three things that are easy to state and easy to get wrong: that the *order* of the edge list changes how quickly the estimates settle but not where they settle; that the closing check is the negative-cycle detector; and that the predecessor array, which normally holds a shortest-path tree, holds a cycle when a negative cycle is present (Section 13.8.8).

Solution to (a). Initialize $d[s] = 0$, all others ∞, all $\pi = \text{NIL}$. Note first the cycles in the graph: $b \to c \to d \to b$ weighs $-2 + 3 + 1 = 2 > 0$, and it is the only cycle, so there is no negative cycle and the algorithm should return TRUE.

Pass 1, relaxing in the listed order:

1. (b, c): $d[b] = \infty$, nothing to relax from.
2. (a, b): $d[a] = \infty$, nothing.
3. (s, a): $0 + 3 = 3 < \infty$, so $d[a] = 3, \pi[a] = s$.
4. (s, b): $0 + 6 = 6 < \infty$, so $d[b] = 6, \pi[b] = s$.
5. (c, d): $d[c] = \infty$, nothing.

6. (a, c): $3 + 5 = 8 < \infty$, so $d[c] = 8, \pi[c] = a$.
7. (d, b): $d[d] = \infty$, nothing.

Three changes. The estimate for d is still ∞ even though c was reached in this pass — because (c, d) was examined *before* (a, c) gave c a finite estimate. The edge order has cost us a pass.

Pass 2:

1. (b, c): $6 + (-2) = 4 < 8$, so $d[c] = 4, \pi[c] = b$.
2. (a, b): $3 + 4 = 7 \not< 6$.
3. (s, a) and 4. (s, b): no change.
4. (c, d): $4 + 3 = 7 < \infty$, so $d[d] = 7, \pi[d] = c$.
5. (a, c): $3 + 5 = 8 \not< 4$.
6. (d, b): $7 + 1 = 8 \not< 6$.

Two changes. *Passes 3 and 4* change nothing: every edge is examined and every test fails. The *closing check* is a fifth sweep that only tests; since pass 4 changed nothing, it cannot find anything either, and the algorithm returns TRUE.

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

Check the answer against the path-relaxation property. The shortest path to d is $s \to b \to c \to d$, of weight $6 - 2 + 3 = 7$, three edges. Its first edge (s, b) is relaxed in pass 1; its second, (b, c), in pass 2; its third, (c, d), also in pass 2, because it comes *after* (b, c) in the list. So the path is fully relaxed, in order, by the end of pass 2 — which is exactly when $d[d]$ became 7. The bound $|V| - 1 = 4$ passes is a worst case over edge orders; this order needed two.

Answer to (a): $\delta(s,\cdot) = (s{:}\,0,\ a{:}\,3,\ b{:}\,6,\ c{:}\,4,\ d{:}\,7)$, predecessor tree $a \leftarrow s, b \leftarrow s, c \leftarrow b, d \leftarrow c$, and the algorithm returns TRUE.

Solution to (b). Now $b \to c \to d \to b$ weighs $-2 + 3 + (-2) = -1 < 0$: a negative cycle, reachable from s through $s \to b$. Shortest-path weights to b, c and d are therefore $-\infty$, and Bellman–Ford must say so by returning FALSE.

Pass 1 is identical to (a), because $d[d] = \infty$ throughout it: $(0,3,6,8,\infty)$.

Pass 2: (b, c) gives $d[c] = 6 - 2 = 4$ ($\pi[c] = b$); (c, d) gives $d[d] = 4 + 3 = 7$ ($\pi[d] = c$); and now (d, b) gives $7 + (-2) = 5 < 6$, so $d[b] = 5, \pi[b] = d$. Three changes.

Pass 3: (b, c): $5 - 2 = 3 < 4$, so $d[c] = 3$; (c, d): $3 + 3 = 6 < 7$, so $d[d] = 6$; (d, b): $6 - 2 = 4 < 5$, so $d[b] = 4$. Three changes — each estimate on the cycle has fallen by exactly 1, the weight of one trip around it.

Pass 4: the same three edges fire again: $d[c] = 2, d[d] = 5, d[b] = 3$.

Closing check: (b, c): $d[b] + w(b, c) = 3 + (-2) = 1 < 2 = d[c]$. An edge can still be relaxed after $|V| - 1$ passes, so the algorithm returns FALSE.

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

Exhibiting the cycle. Detection alone says *that* a negative cycle exists; the predecessor array says *where*. Start at c, the head of the edge that failed the check, and follow π backward $|V| = 5$ times: $c \to b \to d \to c \to b \to d$. We stand at d. Why is five steps enough to be *on* the cycle? A chain of six vertices drawn from only five must repeat one, so after $|V|$ steps the walk has entered the loop it can never leave; the vertex reached is on it. (That the chain from a vertex which can still be improved leads into a cycle, rather than back to s, is the predecessor-subgraph argument of Cormen et al. [1, ch. 22].) Now walk from d until we return to d: $d \to \pi[d] = c \to \pi[c] = b \to \pi[b] = d$. Reversing the order in which we met them gives the cycle in the direction of its edges:

$$b \to c \to d \to b, \qquad w = -2 + 3 + (-2) = -1.$$

Program. **Listing 16.9** runs both parts.

Listing 16.9 Bellman–Ford pass by pass on the graphs of Problem 16.7, with the number of changes in each pass, the closing check, and — when the check fails — the negative cycle recovered from the predecessor array.

```
// Problem 16.7: Bellman-Ford pass by pass, first on a graph with negative edges
// but no negative cycle, then on the same graph with one weight changed so that
// a negative cycle appears -- which the extra pass detects and the predecessor
// array then exhibits.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <climits>
using namespace std;

const int INF = INT_MAX / 4;
struct Edge { int u, v, w; };
const vector<string> nm = {"s", "a", "b", "c", "d"};

void show(const string& tag, const vector<int>& d, const vector<int>& pi) {
    cout << setw(8) << left << tag << right;
    for (size_t v = 0; v < d.size(); ++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";
}

void run(const vector<Edge>& E) {
    int n = (int)nm.size();
    vector<int> d(n, INF), pi(n, -1);
    d[0] = 0;
    cout << setw(8) << left << "pass" << right;
    for (int v = 0; v < n; ++v) cout << setw(8) << ("d/pi " + nm[v]);
    cout << "\n";
    show("init", d, pi);
    for (int pass = 1; pass <= n - 1; ++pass) {
        int changes = 0;
        for (auto& e : E)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) {
                d[e.v] = d[e.u] + e.w; pi[e.v] = e.u; ++changes;
            }
        show(to_string(pass) + " (" + to_string(changes) + ")", d, pi);
    }
    // the extra pass: can anything still be relaxed?
    for (auto& e : E)
        if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) {
            cout << "check: edge " << nm[e.u] << "->" << nm[e.v] << " still relaxes ("
```

Listing 16.9 (Continued) *Bellman–Ford pass by pass on the graphs of Problem 16.7, with the number of changes in each pass, the closing check, and — when the check fails — the negative cycle recovered from the predecessor array.*

```
                << d[e.u] << " + (" << e.w << ") = " << d[e.u] + e.w << " < " << d[e.v]
                << ")  =>  return FALSE\n";
            int x = e.v;
            pi[x] = e.u;                          // record the improving edge
            for (int i = 0; i < n; ++i) x = pi[x]; // walk back n steps: now on the cycle
            vector<int> cyc = {x};
            for (int y = pi[x]; y != x; y = pi[y]) cyc.push_back(y);
            cout << "negative cycle:";
            int total = 0;
            for (int i = (int)cyc.size() - 1; i >= 0; --i) cout << " " << nm[cyc[i]];
            cout << " " << nm[cyc.back()] << "\n";
            for (size_t i = 0; i < cyc.size(); ++i) {
                int to = cyc[i], from = cyc[(i + 1) % cyc.size()];
                for (auto& f : E) if (f.u == from && f.v == to) total += f.w;
            }
            cout << "its weight: " << total << "\n";
            return;
        }
    cout << "check: no edge relaxes  =>  return TRUE (no negative cycle reachable)\n";
}

int main() {
    // The edge list, in the order every pass relaxes it.
    vector<Edge> G = {{2,3,-2}, {1,2,4}, {0,1,3}, {0,2,6},
                      {3,4,3}, {1,3,5}, {4,2,1}};
    cout << "Graph 1: d->b has weight 1 (cycle b->c->d->b weighs -2+3+1 = 2)\n";
    run(G);
    G[6].w = -2;
    cout << "\nGraph 2: d->b has weight -2 (cycle b->c->d->b weighs -2+3-2 = -1)\n";
    run(G);
    return 0;
}
```

It prints:

```
Graph 1: d->b has weight 1 (cycle b->c->d->b weighs -2+3+1 = 2)
pass      d/pi s  d/pi a  d/pi b  d/pi c  d/pi d
init        0/-   inf/-   inf/-   inf/-   inf/-
1 (3)       0/-     3/s     6/s     8/a   inf/-
2 (2)       0/-     3/s     6/s     4/b     7/c
3 (0)       0/-     3/s     6/s     4/b     7/c
4 (0)       0/-     3/s     6/s     4/b     7/c
check: no edge relaxes  =>  return TRUE (no negative cycle reachable)

Graph 2: d->b has weight -2 (cycle b->c->d->b weighs -2+3-2 = -1)
pass      d/pi s  d/pi a  d/pi b  d/pi c  d/pi d
init        0/-   inf/-   inf/-   inf/-   inf/-
1 (3)       0/-     3/s     6/s     8/a   inf/-
2 (3)       0/-     3/s     5/d     4/b     7/c
3 (3)       0/-     3/s     4/d     3/b     6/c
4 (3)       0/-     3/s     3/d     2/b     5/c
check: edge b->c still relaxes (3 + (-2) = 1 < 2)  =>  return FALSE
negative cycle: b c d b
its weight: -1
```

The numbers in parentheses after each pass number are the counts of successful relaxations, and they match the last columns of **Table 16.9** and **Table 16.10**. The zero counts in part (a) also suggest a cheap optimization that Chapter 13 mentioned: **a pass with no change means nothing will ever change again**, because the next pass would examine exactly the same numbers, so the algorithm may stop early. In part (b) no pass is ever quiet — a negative cycle keeps every estimate on it falling forever — which is another way to see why the check must fail.

Check yourself. In part (a), reverse the order of the edge list and run Bellman–Ford again. How many passes change something, and are the values it ends with the same?

Answer. In the order $(d, b, 1), (a, c, 5), (c, d, 3), (s, b, 6), (s, a, 3), (a, b, 4), (b, c, -2)$, pass 1 sets only $d[b] = 6$, $d[a] = 3$ and then $d[c] = 6 - 2 = 4$ (the edges out of a were examined before $d[a]$ was finite). Pass 2 sets $d[d] = 4 + 3 = 7$ via (c, d). Pass 3 changes nothing. So two passes change something, and the values it ends with are the same, (0,3,6,4,7) — as they must be, since the answer does not depend on the order, only the speed of arriving at it does.

16.8.3 Problem 16.8 — shortest paths in a directed acyclic graph

Problem 16.8. A directed acyclic graph on the vertices s, a, b, c, d, e has the edges

$$s \to a\ (6),\ \ s \to b\ (3),\ \ b \to a\ (-2),\ \ b \to c\ (4),\ \ a \to c\ (1),\ \ a \to d\ (5),\ \ c \to d\ (-3),\ \ c \to e\ (2),\ \ d \to e\ (1),\ \ b \to d\ (7),$$

with adjacency lists stored in exactly this order. (a) Find a topological order by depth-first search from s, giving the discovery and finish times. (b) Compute the shortest-path weights and predecessors from s by relaxing in that order, showing d/π after each vertex is processed. (c) Say why no priority queue is needed, and why Dijkstra's algorithm would not be the right tool even though it happens to succeed here.

What it tests. Topological sorting by depth-first search (Chapter 10) combined with relaxation (Chapter 13): the algorithm of Section 13.9, whose correctness rests on the fact that every edge goes *forward* in a topological order. **Pseudocode 16.2** is the procedure.

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)
```

Solution to (a). Run DFS from s, taking neighbors in list order, with a clock that ticks at every discovery and every finish.

- s discovered at time 1. First neighbor a.
- a discovered at 2. First neighbor c.
- c discovered at 3. First neighbor d.
- d discovered at 4. Its only neighbor e.
- e discovered at 5; no neighbors; e finishes at 6.
- Back in d: no more neighbors; d finishes at 7.
- Back in c: next neighbor e is already finished; c finishes at 8.
- Back in a: next neighbor d is finished; a finishes at 9.
- Back in s: next neighbor b is undiscovered; b discovered at 10. Its neighbors a, c, d are all finished; b finishes at 11.
- s finishes at 12.

Listing the vertices by *decreasing* finish time gives the topological order

$$s\ (12),\quad b\ (11),\quad a\ (9),\quad c\ (8),\quad d\ (7),\quad e\ (6).$$

Check that every edge goes forward: $s \to a$, $s \to b$ (positions 1 to 3, 1 to 2); $b \to a$, $b \to c$, $b \to d$ (2 to 3, 4, 5); $a \to c$, $a \to d$ (3 to 4, 5); $c \to d$, $c \to e$ (4 to 5, 6); $d \to e$ (5 to 6). All ten go forward. Note that the alphabetical order $s, a, b, \ldots$ would *not* do: the edge $b \to a$ would go backward.

Solution to (b). Initialize $d[s] = 0$, others ∞. Process the vertices in topological order and relax each out-edge once.

- s $(d = 0)$: $s \to a$: $0 + 6 = 6$, so $d[a] = 6, \pi[a] = s$. $s \to b$: $0 + 3 = 3$, so $d[b] = 3, \pi[b] = s$.
- b $(d = 3)$: $b \to a$: $3 + (-2) = 1 < 6$, so $d[a] = 1, \pi[a] = b$. $b \to c$: $3 + 4 = 7$, so $d[c] = 7, \pi[c] = b$. $b \to d$: $3 + 7 = 10$, so $d[d] = 10, \pi[d] = b$.
- a $(d = 1)$: $a \to c$: $1 + 1 = 2 < 7$, so $d[c] = 2$, $\pi[c] = a$. $a \to d$: $1 + 5 = 6 < 10$, so $d[d] = 6$, $\pi[d] = a$.
- c $(d = 2)$: $c \to d$: $2 + (-3) = -1 < 6$, so $d[d] = -1$, $\pi[d] = c$. $c \to e$: $2 + 2 = 4$, so $d[e] = 4$, $\pi[e] = c$.
- d $(d = -1)$: $d \to e$: $-1 + 1 = 0 < 4$, so $d[e] = 0, \pi[e] = d$.
- e $(d = 0)$: no out-edges.

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	$0/-$	$1/b$	$3/s$	$2/a$	$-1/c$	$0/d$

Answer to (b): $\delta(s,\cdot) = (s{:}\,0,\ a{:}\,1,\ b{:}\,3,\ c{:}\,2,\ d{:}\,-1,\ e{:}\,0)$, and the shortest path to e is $s \to b \to a \to c \to d \to e$ with weight $3 - 2 + 1 - 3 + 1 = 0$.

Solution to (c). When u is processed, every edge *into* u comes from a vertex earlier in the topological order, which has already been processed and whose out-edges — including the one into u — have already been relaxed. So by the path-relaxation property, $d[u]$ is final at that moment, whatever the signs of the weights. The order in which vertices become final is fixed in advance by the topological sort, so there is nothing to *choose*, and a priority queue, whose only job is choosing, is unnecessary. Total cost: $\Theta(V + E)$ for the sort plus $\Theta(V + E)$ for the relaxations, here $6 + 10$ steps of each.

Dijkstra, on this graph, happens to give the right answer: it extracts $s\,(0)$, $b\,(3)$, $a\,(1)$, $c\,(2)$, $d\,(-1)$, $e\,(0)$ and no finalized vertex is ever improved later. But that is luck, not a theorem: its correctness proof (Section 13.7.8) needs non-negative weights, and the extraction values $0,3,1,2,-1,0$ are not even increasing, which is exactly the invariant that proof relies on. A program that is right on one input with no proof behind it is the guess that Chapter 8 warned against.

Program. **Listing 16.10** performs both steps.

Listing 16.10 Shortest paths in the DAG of Problem 16.8: a depth-first topological sort, then one relaxation per edge in topological order, with d/π printed after each vertex.

```
// Problem 16.8: single-source shortest paths in a DAG with negative edges.
// Step 1 is a topological sort by depth-first search (Chapter 10); step 2
// relaxes the out-edges of each vertex once, in that order (Chapter 13).
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;

const int INF = INT_MAX / 4;
const vector<string> nm = {"s", "a", "b", "c", "d", "e"};
vector<vector<pair<int,int>>> adj(6);
vector<int> color(6, 0), order;                  // 0 white, 1 gray, 2 black

void dfs(int u) {
    color[u] = 1;
    for (auto [v, w] : adj[u]) if (color[v] == 0) dfs(v);
    color[u] = 2;
    order.push_back(u);                          // appended when u finishes
}

int main() {
    auto add = [](int u, int v, int w) { adj[u].push_back({v, w}); };
    add(0,1,6); add(0,2,3); add(2,1,-2); add(2,3,4); add(1,3,1);
    add(1,4,5); add(3,4,-3); add(3,5,2); add(4,5,1); add(2,4,7);
    int n = (int)nm.size();

    for (int u = 0; u < n; ++u) if (color[u] == 0) dfs(u);
    reverse(order.begin(), order.end());         // decreasing finish time
    cout << "topological order:";
    for (int v : order) cout << " " << nm[v];
    cout << "\n\n";

    vector<int> d(n, INF), pi(n, -1);
    d[0] = 0;
    cout << "process";
    for (int v = 0; v < n; ++v) cout << setw(8) << ("d/pi " + nm[v]);
    cout << "\n";
    for (int u : order) {
        if (d[u] < INF)
            for (auto [v, w] : adj[u])
                if (d[u] + w < d[v]) { d[v] = d[u] + w; pi[v] = u; }
        cout << setw(7) << nm[u];
        for (int v = 0; v < n; ++v) {
```

Listing 16.10 (Continued) *Shortest paths in the DAG of Problem 16.8: a depth-first topological sort, then one relaxation per edge in topological order, with d/\pi printed after each vertex.*

```
            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 << "\nshortest path to e:";
    vector<int> path;
    for (int x = 5; x != -1; x = pi[x]) path.push_back(x);
    for (int i = (int)path.size() - 1; i >= 0; --i) cout << " " << nm[path[i]];
    cout << "   weight " << d[5] << "\n";
    return 0;
}
```

It prints:

```
topological order: s b a c d e

process  d/pi s  d/pi a  d/pi b  d/pi c  d/pi d  d/pi e
      s     0/-     6/s     3/s   inf/-   inf/-   inf/-
      b     0/-     1/b     3/s     7/b    10/b   inf/-
      a     0/-     1/b     3/s     2/a     6/a   inf/-
      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

shortest path to e: s b a c d e   weight 0
```

The topological order and every row agree with the hand computation of **Table 16.11**.

Check yourself. Change the weight of $b \to a$ from -2 to $+2$. Recompute d/π for every vertex.

Answer. a: $5/b$, b: $3/s$, c: $6/a$, d: $3/c$, e: $4/d$. (At a: $\min(6,3+2) = 5$. At c: $\min(3+4,5+1) = 6$. At d: $\min(3+7,5+5,6-3) = 3$ — the tie between 10 and 10 is resolved by the strict test in favor of b, and then c beats both. At e: $\min(6+2,3+1) = 4$.)

16.9 Worked Problems on Amortized Analysis

Chapter 14 introduced three methods for one question: *what does a sequence of n operations cost in total, when a few of them are expensive?* The two problems of this section apply all three methods to the same data structure, side by side, so that you can see that they are three bookkeepings of one fact rather than three different theories. Both problems use the potential-method identity of Section 14.6, which is worth writing once more before we start. If operation i has actual cost c_i and takes the structure from state D_{i-1} to state D_i, its **amortized cost** with respect to a potential function Φ is

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}), \tag{16.6}$$

and summing over $i = 1, \ldots, n$, the potentials telescope:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0), \qquad \text{so} \qquad \sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i \ \text{ whenever } \ \Phi(D_n) \ge \Phi(D_0). \tag{16.7}$$

The condition $\Phi(D_n) \ge \Phi(D_0)$ must hold for *every* n, because we want the bound for every prefix of the sequence; that is why one normally takes $\Phi(D_0) = 0$ and proves $\Phi \ge 0$ throughout (Section 14.6.3).

16.9.1 Problem 16.9 — the binary counter, three ways and by induction

Problem 16.9. A k-bit binary counter $A[0..k-1]$ starts at zero, and the procedure `INCREMENT` of **Pseudocode 16.3** is called n times. Flipping one bit costs 1. Show that the total cost of the n calls is less than $2n$, so that the amortized cost per call is $O(1)$, in four ways: (a) by the aggregate method, (b) by the accounting method, (c) by the potential method, and (d) by induction on n. Tabulate the first eight increments for all of them.

Pseudocode 16.3 `INCREMENT` on a k-bit binary counter: clear the trailing 1 bits, then set the first 0 bit.

```
INCREMENT(A, k)
i = 0
while i < k and A[i] == 1
    A[i] = 0
    i = i + 1
if i < k
    A[i] = 1
```

What it tests. All three methods of Chapter 14 (Sections 14.3.3, 14.5.4 and 14.6.5) on the structure they were first taught with, plus the observation that the potential method is, underneath, a strengthened induction hypothesis — the same move as the substitution proof of Chapter 4.

Why the naive bound is useless. A single `INCREMENT` can flip all k bits (from $0111\ldots1$ to $1000\ldots0$), so n calls cost at most nk. True, and useless: it charges every call as if it were the worst one. **Figure 16.9** shows the first sixteen increments of an 8-bit counter with the bits that flip shaded; almost every call flips one or two bits.

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 $2 \times 16 = 32$.

Solution (a): the aggregate method. Count by *bit* rather than by *call*. Bit $A[0]$ flips on every call: n times. Bit $A[1]$ flips on every second call: $\lfloor n/2 \rfloor$ times. In general bit $A[i]$ flips exactly when the low i bits are all 1 before the call, which happens once in every 2^i calls, so it flips $\lfloor n/2^i \rfloor$ times. Therefore the total cost $C(n)$ is

$$C(n) \;=\; \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \;<\; \sum_{i=0}^{\infty} \frac{n}{2^i} \;=\; n \cdot \frac{1}{1-1/2} \;=\; 2n. \tag{16.8}$$

The amortized cost per call is $C(n)/n < 2$. Check Equation (16.8) at $n = 16$: $16 + 8 + 4 + 2 + 1 = 31 < 32$, exactly the last entry of **Figure 16.9**.

Solution (b): the accounting method. Charge every `INCREMENT` a fee of **2**. When a bit is set from 0 to 1 (line 6, which happens at most once per call), spend 1 of the fee on that flip and leave the other 1 *on the bit* as credit. When a bit is reset from 1 to 0 (line 3), pay for it with the credit sitting on it. The invariant to prove is that the bank never goes negative, and it holds for a simple reason: **every 1 bit carries exactly one unit of credit**, so the bank equals the number of 1 bits, which is never negative. The resets in lines 2–4 are therefore always prepaid, each call pays at most 2 out of pocket, and n calls cost at most $2n$.

Solution (c): the potential method. Let $\Phi(D_i) = b_i$, the number of 1 bits after the i-th call. Then $\Phi(D_0) = 0$ and $\Phi \geq 0$ always. Suppose call i resets t_i bits in lines 2–4. Then its actual cost is at most $t_i + 1$ (t_i resets plus at most one set), and the number of 1 bits changes by at most $-t_i + 1$:

$$b_i \;\leq\; b_{i-1} - t_i + 1.$$

(Equality holds unless the counter overflows, when all k bits are reset and none is set.) By Equation (16.6),

$$\hat{c}_i \;=\; c_i + b_i - b_{i-1} \;\le\; (t_i + 1) + (1 - t_i) \;=\; 2.$$

Every amortized cost is at most 2, the potential starts at 0 and never goes negative, so by Equation (16.7) the total actual cost is at most $2n$.

Take one expensive call and check the arithmetic. Call 8 takes 0111 to 1000: $t_8 = 3$ resets and one set, so $c_8 = 4$; the potential falls from $b_7 = 3$ to $b_8 = 1$; the amortized cost is $\hat{c}_8 = 4 + 1 - 3 = 2$. Call 16 takes 01111 to 10000: $c_{16} = 5$, Φ falls from 4 to 1, and $\hat{c}_{16} = 5 + 1 - 4 = 2$. The expensive calls are paid for by the potential that the cheap calls built up.

Solution (d): by induction. The claim $C(n) < 2n$ does not carry itself through an induction: knowing $C(n-1) < 2(n-1)$ says nothing useful about call n, which may cost as much as k. The repair — the same strengthening trick as the substitution proofs of Chapter 4 — is to carry a second quantity along. Let $v(n)$ be the number of 1 bits after n calls, and prove

$$P(n): \qquad C(n) + v(n) \le 2n.$$

Base, $n = 0$. $C(0) = 0$, $v(0) = 0$, and $0 \le 0$.

Step. Assume $C(n-1) + v(n-1) \le 2(n-1)$. Call n turns t trailing 1s into 0s and one 0 into a 1 (ignore overflow, which only helps), so it costs $t + 1$ and changes the number of 1 bits by $-t + 1$. Therefore

$$\begin{aligned} C(n) + v(n) \;&=\; (C(n-1) + t + 1) + (v(n-1) - t + 1) \\ &=\; C(n-1) + v(n-1) + 2 \;\le\; 2(n-1) + 2 \;=\; 2n. \end{aligned}$$

Since $v(n) \ge 1$ for $n \ge 1$ (the counter is not zero after a non-overflowing increment), $C(n) \le 2n - 1 < 2n$. ■

Look at what the strengthened quantity is: $C(n) + v(n)$ is the actual total plus the potential of (c). **The potential method is an induction with a strengthened hypothesis, written as bookkeeping.**

The first eight calls, all methods at once. **Table 16.12** computes every quantity by hand. The column *bank* is the accounting balance after the call (fees paid minus actual costs), and the column $\hat{c}_i$ is Equation (16.6) with Φ = the number of 1 bits.

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	$4 + 1 - 3 = 2$

The equality of the *bank* and Φ_i columns is not a coincidence: the credit of the accounting method and the potential of the potential method are the same number, one stored on the bits and one computed from them.

Program. Listing 16.11 runs sixteen calls with all three ledgers, and then checks Equation (16.8) for larger n.

Listing 16.11 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.

```
// 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;
}
```

It prints:

```
  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
```

The first eight rows are **Table 16.12**; the total of 31 at $i = 16$ is the last entry of **Figure 16.9**; and at $n = 1000$ the total is $1994 = \sum_i \lfloor 1000/2^i \rfloor$, an amortized 1.994 per call.

Check yourself. Suppose flipping bit i costs 2^i instead of 1 (high bits are expensive to write). Using the aggregate method, find the total cost of $n = 2^m$ increments from zero, and the amortized cost per call.

Answer. Bit i flips $\lfloor n/2^i \rfloor = 2^{m-i}$ times at cost 2^i each, contributing exactly $2^m = n$ for each $i = 0,1,\ldots,m$. The total is $(m+1)\,n = n(\lg n + 1)$, so the amortized cost is $\Theta(\lg n)$ per call — no longer constant. Amortization spreads cost; it does not make it disappear.

16.9.2 Problem 16.10 — the dynamic table, three ways, and a tempting wrong potential

Problem 16.10. A dynamic table starts empty with $size = 0$. `TABLE-INSERT` (**Pseudocode 16.4**) writes one item; if the table is full it first allocates a new table of twice the size (size 1 when the old size was 0), copies every item across, and frees the old one. Writing or copying one item costs 1. (a) Give the actual cost c_i of the i-th insertion. (b) Show that n insertions cost less than $3n$ by the aggregate method, (c) by the accounting method with a fee of 3, and (d) by the potential method with

$$\Phi(T) \;=\; 2 \cdot num[T] - size[T]. \tag{16.9}$$

(e) Tabulate the first nine insertions. (f) Explain what goes wrong with the potential $\Phi(T) = num[T] - size[T]$.

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)
if T.size == 0
    allocate T.table with 1 slot;  T.size = 1
if T.num == T.size
    allocate new-table with 2 * T.size slots
    insert all items in T.table into new-table
    free T.table;  T.table = new-table;  T.size = 2 * T.size
insert x into T.table
T.num = T.num + 1
```

What it tests. The dynamic table of Section 14.7, analyzed by the three methods of Sections 14.7.3 to 14.7.5, and the two requirements a potential function must meet.

Solution (a). The i-th insertion finds $num = i - 1$ items in the table. An expansion happens exactly when the table is full, that is when $i - 1$ equals the current size, and the sizes run 1,2,4,8, ... So

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \quad (i-1 \text{ copies, plus 1 write}), \\ 1 & \text{otherwise.} \end{cases}$$

(The very first insertion allocates a table of size 1 and writes one item: $c_1 = 1$.) The expensive insertions are $i = 2,3,5,9,17, \ldots$, costing 2,3,5,9,17, ...

Solution (b): aggregate. Every insertion pays 1 for its own write, which totals n. On top of that, the copies at the expansions total $1 + 2 + 4 + \cdots + 2^j$ where $2^j \le n - 1$, and a geometric sum is less than twice its largest term:

$$\sum_{i=1}^{n} c_i \;\le\; n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j \;<\; n + 2n \;=\; 3n.$$

For $n = 17$: the writes cost 17, the copies at $i = 2,3,5,9,17$ cost $1 + 2 + 4 + 8 + 16 = 31$, and the total 48 is indeed below $3 \times 17 = 51$.

Solution (c): accounting. Charge every insertion **3**. Spend 1 on writing the item. Keep 2 as credit, and picture it as follows: 1 unit pays for moving *this item* at the next expansion, and 1 unit pays for moving *one old item* that was moved at the previous expansion and has already spent its own credit. Why does that balance? Just after an expansion to size $2m$ the table holds $m + 1$ items, the m old ones with no credit left. Before the next expansion, $m - 1$ further items arrive, each leaving 2 units, plus the one that arrived with the expansion: the bank holds $2m$ units when the table of $2m$ items is full — exactly enough to move all $2m$ of them. So the bank never goes negative, and n insertions cost at most $3n$.

Solution (d): potential. Check the two requirements for Equation (16.9) first. The empty table has $num = size = 0$, so $\Phi(T_0) = 0$. After the first insertion the table is always at least half full — it is exactly half full plus one immediately after an expansion, since the table doubled from full and one item was added, and insertions only raise num — so $2\,num \ge size$ and $\Phi \ge 0$. Now compute $\hat{c}_i$ by Equation (16.6), writing num_i, $size_i$ for the values after insertion i.

Case 1: no expansion. Then $size_i = size_{i-1}$, $num_i = num_{i-1} + 1$ and $c_i = 1$:

$$\hat{c}_i \;=\; 1 + (2\,num_i - size_i) - (2(num_i - 1) - size_i) \;=\; 1 + 2 \;=\; 3.$$

Case 2: expansion. Then $size_{i-1} = num_{i-1} = num_i - 1$, $size_i = 2\,size_{i-1} = 2(num_i - 1)$, and $c_i = num_i$ ($num_i - 1$ copies plus 1 write):

$$\begin{aligned}\hat{c}_i &= num_i + (2\,num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\ &= num_i + 2 - (num_i - 1) = 3.\end{aligned}$$

(For $i = 1$, where the size goes from 0 to 1, $\hat{c}_1 = 1 + (2 - 1) - 0 = 2$.) Every amortized cost is at most 3, Φ never falls below $\Phi(T_0) = 0$, so by Equation (16.7) the total actual cost of n insertions is at most $3n$. The potential is the savings account: it climbs by 2 with every cheap insertion and is spent all at once on the copy.

Solution (e). **Table 16.13** carries out every one of these computations for the first nine insertions.

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	$\sum c$	$3i$	Bank	$\Phi_i = 2num_i - size_i$	c_i
1	1	1	1	1	3	2	$2 - 1 = 1$	$1 + 1 - 0 = 2$
2	2	2	2	3	6	3	$4 - 2 = 2$	$2 + 2 - 1 = 3$
3	4	3	3	6	9	3	$6 - 4 = 2$	$3 + 2 - 2 = 3$
4	4	4	1	7	12	5	$8 - 4 = 4$	$1 + 4 - 2 = 3$
5	8	5	5	12	15	3	$10 - 8 = 2$	$5 + 2 - 4 = 3$
6	8	6	1	13	18	5	$12 - 8 = 4$	$1 + 4 - 2 = 3$
7	8	7	1	14	21	7	$14 - 8 = 6$	$1 + 6 - 4 = 3$
8	8	8	1	15	24	9	$16 - 8 = 8$	$1 + 8 - 6 = 3$
9	16	9	9	24	27	3	$18 - 16 = 2$	$9 + 2 - 8 = 3$

The bank is always $\Phi_i + 1$: the one extra unit is the change left over from the first insertion, whose amortized cost was 2 but which was charged 3.

Solution (f): a tempting but wrong potential. Why $2\,num - size$ and not something simpler? Try $\Phi(T) = num - size$, which is also 0 on the empty table and looks just as natural. Compute: in Case 1, $\Delta\Phi = +1$ and $\hat{c}_i = 1 + 1 = 2$; in Case 2, $\Delta\Phi = (num_i - 2(num_i - 1)) - 0 = 2 - num_i$ and $\hat{c}_i = num_i + 2 - num_i = 2$. Every amortized cost is at most 2! If that were a valid analysis, n insertions would cost at most $2n$ — but at $n = 17$ we counted an actual total of 48, far above $2 \times 17 = 34$. The "proof" fails because this Φ is **negative**: a table that is less than full has $num < size$, and at $n = 17$, $\Phi = 17 - 32 = -15$. By Equation (16.7), $\sum c_i = \sum \hat{c}_i - \Phi(D_n) + \Phi(D_0) = 33 + 15 = 48$ — the sum of the amortized costs, 33, understates the truth by exactly the debt the potential has run up. A potential that may go below its starting value borrows against the future, and the bound it gives is worthless. **Always check $\Phi(D_i) \geq \Phi(D_0)$ before believing an amortized bound.**

A second tempting choice, $\Phi(T) = num$, does satisfy $\Phi \geq 0$ but fails the other way: in Case 2, $\hat{c}_i = num_i + 1$, which is not a constant, so it proves nothing better than the naive bound. A good potential must be large *exactly when an expensive operation is imminent* — full table, large Φ — and small just after it. $2\,num - size$ is the simplest function with that shape: it rises from 2 to $size$ as the table fills.

Program. Listing 16.12 prints all three ledgers for seventeen insertions and the aggregate totals for larger n.

Listing 16.12 The doubling dynamic table of Problem 16.10, with the actual cost, the running total against $3i$, the accounting bank, the potential $2\,num - size$ and the amortized cost of every insertion.

```
// Problem 16.10: TABLE-INSERT into a table that doubles when full, analyzed by
// the aggregate, accounting and potential methods side by side.
//   real cost c_i  = 1, plus (i-1) copies when the table was full
//   accounting     : charge 3 per insertion; bank = charges - real costs
//   potential      : Phi = 2*num - size (and Phi = 0 for the empty table)
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    long long size = 0, num = 0, total = 0, bank = 0, phiPrev = 0;
    cout << "  i  size  num  c_i  total   3i  bank  Phi  amortized\n";
    for (int i = 1; i <= 17; ++i) {
        long long cost = 1;                        // writing the new item
        if (num == size) {                         // full: allocate and copy
            cost += num;                           // num items are moved
            size = (size == 0) ? 1 : 2 * size;
        }
        ++num;
        total += cost;
        bank += 3 - cost;
        long long phi = 2 * num - size;
        long long amort = cost + phi - phiPrev;
        phiPrev = phi;
        cout << setw(3) << i << setw(6) << size << setw(5) << num << setw(5) << cost
             << setw(7) << total << setw(5) << 3 * i << setw(6) << bank
             << setw(5) << phi << setw(8) << amort << "\n";
    }
    for (long long n : {17LL, 1000LL, 1000000LL}) {
        long long s = 0, sz = 0, c = 0;
        for (long long k = 1; k <= n; ++k) {
            c += 1;
            if (s == sz) { c += s; sz = sz ? 2 * sz : 1; }
            ++s;
        }
        cout << "n = " << setw(7) << n << ":  total cost = " << setw(7) << c
             << "  <  3n = " << 3 * n << "\n";
    }
    return 0;
}
```

It prints:

```
  i  size  num  c_i  total   3i  bank  Phi  amortized
  1     1    1    1      1    3     2    1          2
  2     2    2    2      3    6     3    2          3
  3     4    3    3      6    9     3    2          3
  4     4    4    1      7   12     5    4          3
  5     8    5    5     12   15     3    2          3
  6     8    6    1     13   18     5    4          3
  7     8    7    1     14   21     7    6          3
  8     8    8    1     15   24     9    8          3
  9    16    9    9     24   27     3    2          3
 10    16   10    1     25   30     5    4          3
 11    16   11    1     26   33     7    6          3
 12    16   12    1     27   36     9    8          3
 13    16   13    1     28   39    11   10          3
 14    16   14    1     29   42    13   12          3
 15    16   15    1     30   45    15   14          3
 16    16   16    1     31   48    17   16          3
 17    32   17   17     48   51     3    2          3
n =      17:  total cost =      48  <  3n = 51
n =    1000:  total cost =    2023  <  3n = 3000
n = 1000000:  total cost = 2048575  <  3n = 3000000
```

The first nine rows are **Table 16.13**, and the aggregate totals — 48 for $n = 17$, 2023 for $n = 1000$, 2048575 for $n = 10^6$ — all lie below $3n$.

Check yourself. Suppose instead that a full table grows by adding a fixed **4** slots (sizes 4,8,12, …), starting from size 0. Find the total cost of 17 insertions, and show by the aggregate method that n insertions cost $\Theta(n^2)$.

Answer. Expansions happen when $num = 0,4,8,12,16$ and copy 0,4,8,12,16 items: 40 copies plus 17 writes, total 57. In general, with $m = \lfloor (n-1)/4 \rfloor$ expansions after the first, the copies total $4(1 + 2 + \cdots + m) = 2m(m+1) \approx n^2/8$, so the total is $\Theta(n^2)$ and the amortized cost per insertion is $\Theta(n)$. (For $n = 1000$: $m = 249$, total $1000 + 2 \cdot 249 \cdot 250 = 125{,}500$.) Growing by a *factor* is what makes the table cheap; growing by a *constant* does not.

16.10 Worked Problems on All-Pairs Shortest Paths

Chapter 15 asked for the shortest-path weight between *every* ordered pair of vertices, and answered with two algorithms that could hardly be more different in spirit: Floyd–Warshall, a three-line dynamic program over matrices, and Johnson's algorithm, a reduction that changes the weights so that Dijkstra may be used. The two problems of this section run both, by hand, on the same four-vertex graph with a negative edge, and check that they agree.

The graph, used in both problems, has vertices 1,2,3,4 and the seven edges

$$1 \to 2\ (4), \quad 1 \to 4\ (5), \quad 2 \to 3\ (-2), \quad 3 \to 1\ (3), \quad 3 \to 4\ (2), \quad 4 \to 2\ (1), \quad 4 \to 3\ (6).$$

It has five cycles: $1 \to 2 \to 3 \to 1$ of weight $4 - 2 + 3 = 5$, $2 \to 3 \to 4 \to 2$ of weight $-2 + 2 + 1 = 1$, $3 \to 4 \to 3$ of weight $2 + 6 = 8$, $1 \to 4 \to 2 \to 3 \to 1$ of weight $5 + 1 - 2 + 3 = 7$, and $1 \to 4 \to 3 \to 1$ of weight $5 + 6 + 3 = 14$. All are positive, so shortest paths are well defined even though one edge is negative.

16.10.1 Problem 16.11 — Floyd–Warshall, every matrix computed cell by cell

Problem 16.11. Run the Floyd–Warshall algorithm on the graph above. Give the matrices $D^{(0)}, D^{(1)}, D^{(2)}, D^{(3)}, D^{(4)}$ and the predecessor matrices $\Pi^{(0)}, \dots, \Pi^{(4)}$, showing for each round k every comparison that could change an entry. Then read the shortest paths $1 \rightsquigarrow 4$, $4 \rightsquigarrow 1$ and $3 \rightsquigarrow 2$ out of $\Pi^{(4)}$, and say how the algorithm would reveal a negative cycle.

What it tests. The Floyd–Warshall recurrence of Section 15.6. Let $d_{ij}^{(k)}$ be the weight of a shortest path from i to j all of whose *interior* vertices lie in $\{1, \dots, k\}$. Such a path either does not use k at all, or passes through k exactly once and splits there into two pieces whose interiors lie in $\{1, \dots, k-1\}$. Hence

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1, \end{cases} \tag{16.10}$$

where w_{ij} is the edge weight, 0 on the diagonal and ∞ where there is no edge. The predecessor rule goes with it: when the second argument wins, $\pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$ — the last hop into j is the last hop of the route from k to j.

Solution. *Round 0.* $D^{(0)}$ is the weight matrix, and $\Pi^{(0)}$ has $\pi_{ij} = i$ wherever there is an edge $i \to j$, and NIL elsewhere. **Table 16.14** shows both.

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

A fact that saves most of the work: in round k, an entry d_{ij} can only change if **both** d_{ik} (column k) and d_{kj} (row k) are finite, and never when $i = k$ or $j = k$ (going through k to reach k adds nothing). So in each round we need only look at the finite entries of column k and row k.

Round 1 (may paths pass through vertex 1?). Column 1, off the diagonal: $d_{21} = \infty$, $d_{31} = 3$, $d_{41} = \infty$. Row 1: $d_{12} = 4$, $d_{13} = \infty$, $d_{14} = 5$. Only $i = 3$ has a finite entry, and only $j = 2,4$:

- d_{32}: $\min(\infty,\ 3 + 4) = 7$. **Changed**; $\pi_{32} = \pi_{12} = 1$. (The route $3 \to 1 \to 2$.)
- d_{34}: $\min(2,\ 3 + 5 = 8) = 2$. No change.

Round 2 (through vertices 1 and 2). Column 2: $d_{12} = 4$, $d_{32} = 7$, $d_{42} = 1$. Row 2: $d_{21} = \infty$, $d_{23} = -2$, $d_{24} = \infty$. Only $j = 3$:

- d_{13}: $\min(\infty,\ 4 + (-2)) = 2$. **Changed**; $\pi_{13} = \pi_{23} = 2$.
- d_{33}: $\min(0,\ 7 + (-2) = 5) = 0$. No change — the diagonal stays 0, as it must without a negative cycle.
- d_{43}: $\min(6,\ 1 + (-2)) = -1$. **Changed**; $\pi_{43} = \pi_{23} = 2$. The negative edge has made the route $4 \to 2 \to 3$ cheaper than the direct edge of weight 6.

Round 3 (through 1, 2, 3). Column 3: $d_{13} = 2$, $d_{23} = -2$, $d_{43} = -1$. Row 3: $d_{31} = 3$, $d_{32} = 7$, $d_{34} = 2$. Now all nine combinations are finite:

- $i = 1$: $d_{11} = \min(0,\ 2+3) = 0$; $d_{12} = \min(4,\ 2+7) = 4$; $d_{14} = \min(5,\ 2+2) = 4$, **changed**, $\pi_{14} = \pi_{34} = 3$.
- $i = 2$: $d_{21} = \min(\infty,\ -2+3) = 1$, **changed**, $\pi_{21} = \pi_{31} = 3$; $d_{22} = \min(0,\ -2+7) = 0$; $d_{24} = \min(\infty,\ -2+2) = 0$, **changed**, $\pi_{24} = \pi_{34} = 3$.
- $i = 4$: $d_{41} = \min(\infty,\ -1+3) = 2$, **changed**, $\pi_{41} = \pi_{31} = 3$; $d_{42} = \min(1,\ -1+7) = 1$; $d_{44} = \min(0,\ -1+2) = 0$.

Round 4 (all vertices allowed). Column 4: $d_{14} = 4$, $d_{24} = 0$, $d_{34} = 2$. Row 4: $d_{41} = 2$, $d_{42} = 1$, $d_{43} = -1$:

- $i = 1$: $d_{11} = \min(0,6) = 0$; $d_{12} = \min(4,\ 4+1=5) = 4$; $d_{13} = \min(2,\ 4-1=3) = 2$.
- $i = 2$: $d_{21} = \min(1,\ 0+2) = 1$; $d_{22} = \min(0,\ 1) = 0$; $d_{23} = \min(-2,\ 0-1=-1) = -2$.
- $i = 3$: $d_{31} = \min(3,\ 2+2=4) = 3$; $d_{32} = \min(7,\ 2+1) = 3$, **changed**, $\pi_{32} = \pi_{42} = 4$; $d_{33} = \min(0,\ 2-1=1) = 0$.

Table 16.15 collects the eight changes of the whole run.

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

Answer. The final matrices are

$$D^{(4)} = \begin{pmatrix} 0 & 4 & 2 & 4 \\ 1 & 0 & -2 & 0 \\ 3 & 3 & 0 & 2 \\ 2 & 1 & -1 & 0 \end{pmatrix}, \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 2 & 3 \\ 3 & \text{NIL} & 2 & 3 \\ 3 & 4 & \text{NIL} & 3 \\ 3 & 4 & 2 & \text{NIL} \end{pmatrix}.$$

Reading paths out of $\Pi^{(4)}$ works backward from the destination, using row i throughout:

- $1 \rightsquigarrow 4$: $\pi_{14} = 3, \pi_{13} = 2, \pi_{12} = 1$. The path is $1 \to 2 \to 3 \to 4$, weight $4 - 2 + 2 = 4 = d_{14}$.
- $4 \rightsquigarrow 1$: $\pi_{41} = 3, \pi_{43} = 2, \pi_{42} = 4$. The path is $4 \to 2 \to 3 \to 1$, weight $1 - 2 + 3 = 2 = d_{41}$.
- $3 \rightsquigarrow 2$: $\pi_{32} = 4$, $\pi_{34} = 3$. The path is $3 \to 4 \to 2$, weight $2 + 1 = 3 = d_{32}$. (The round-1 route $3 \to 1 \to 2$ of weight 7 was correctly abandoned.)

Negative cycles. A negative cycle through vertex i would, by the round in which all its vertices are allowed, give a route from i back to i of negative weight, so it would make $d_{ii} < 0$. Floyd–Warshall therefore detects a negative cycle for free: **after the last round, look at the diagonal** (Section 15.6.14). Here every diagonal entry is 0, so there is none.

Program. Listing 16.13 prints every matrix, with a star on each entry that changed in that round.

Listing 16.13 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.

```
// 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 16.13 (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;
}
```

It prints:

```
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 are exactly the eight rows of **Table 16.15**. The running time is $\Theta(n^3)$: $4^3 = 64$ comparisons here, of which the hand solution skipped most by the "row k and column k must be finite" observation.

Check yourself. Change the weight of $4 \to 2$ from 1 to -1. Which cycle becomes negative, and what does the diagonal of $D^{(4)}$ show?

Answer. The cycle $2 \to 3 \to 4 \to 2$ now weighs $-2 + 2 - 1 = -1$. Running the algorithm gives $D^{(4)}$ with diagonal $(0, -1, -1, -2)$: entries 2, 3 and 4 are negative, which flags the cycle, while $d_{11} = 0$ because vertex 1 is not on it. The entry $d_{44} = -2$ is the weight of going round the cycle *twice*, which is a warning of its own: once a negative cycle exists, the numbers in D are no longer shortest-path weights of anything, and only their signs on the diagonal may be trusted.

16.10.2 Problem 16.12 — Johnson's reweighting, carried out in full

Problem 16.12. Solve the all-pairs problem for the same graph by Johnson's algorithm. (a) Add a new vertex q with an edge of weight 0 to every vertex, run Bellman–Ford from q, and give the potentials $h(v) = \delta(q, v)$. (b) Compute every reweighted edge $w'(u, v)$ and show that none is negative. (c) Run Dijkstra from each vertex on the reweighted graph, and convert the results back to true distances. (d) Show that reweighting cannot change which path is shortest, and compare with Problem 16.11.

What it tests. Section 15.8: the reweighting

$$w'(u, v) \;=\; w(u, v) + h(u) - h(v) \tag{16.11}$$

with h chosen as shortest-path weights from an auxiliary vertex, its two properties — it preserves shortest paths, and it makes every weight non-negative — and the bookkeeping that turns Dijkstra's answers on w' back into answers on w.

Solution to (a). The new vertex q has edges $q \to 1, q \to 2, q \to 3, q \to 4$, all of weight 0; nothing enters q, so no new cycle is created and the new graph has a negative cycle only if the old one did. Run Bellman–Ford from q on the five vertices, relaxing the four q-edges first and then the seven original edges in the order listed.

Pass 1. The q-edges set $h(1) = h(2) = h(3) = h(4) = 0$. Then: $1 \to 2$: $0 + 4 \not< 0$; $1 \to 4$: $0 + 5 \not< 0$; $2 \to 3$: $0 + (-2) = -2 < 0$, so $h(3) = -2$; $3 \to 1$: $-2 + 3 = 1 \not< 0$; $3 \to 4$: $-2 + 2 = 0 \not< 0$; $4 \to 2$: $0 + 1 \not< 0$; $4 \to 3$: $0 + 6 \not< -2$.

Passes 2 to 4 change nothing, and the closing check finds no improvable edge, so there is no negative cycle. The potentials are

$$h(1) = 0, \qquad h(2) = 0, \qquad h(3) = -2, \qquad h(4) = 0.$$

Each $h(v)$ is the weight of the cheapest way to reach v starting anywhere (because q can step to any vertex for free). Only vertex 3 can be reached more cheaply than for free, through the negative edge $2 \to 3$.

Solution to (b). Apply Equation (16.11) to every edge. **Table 16.16** shows the arithmetic.

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	$6+0+2=8$

Every w' is non-negative, and not by luck. Because $h(v) = \delta(q, v)$, the triangle inequality of Chapter 13 gives $h(v) \le h(u) + w(u, v)$ for every edge, which rearranges to $w(u, v) + h(u) - h(v) \ge 0$. Notice also *which* edges came out at exactly 0: (2,3) and (3,4) — the edges on shortest paths from q, where the triangle inequality is tight.

Solution to (c). With non-negative weights, Dijkstra is legal. Run it from each source u to get $\hat{d}(u, v)$, the distance under w'. Here is the run from source 4, in full:

- Extract 4 ($\hat{d} = 0$): relax $4 \to 2$ to $0 + 1 = 1$, and $4 \to 3$ to $0 + 8 = 8$.
- Extract 2 ($\hat{d} = 1$): relax $2 \to 3$: $1 + 0 = 1 < 8$, so $\hat{d}(4,3) = 1$.
- Extract 3 ($\hat{d} = 1$): relax $3 \to 1$ to $1 + 1 = 2$; $3 \to 4$: $1 + 0 \not< 0$.
- Extract 1 ($\hat{d} = 2$): $1 \to 2$: $2 + 4 \not< 1$; $1 \to 4$: $2 + 5 \not< 0$.

So $\hat{d}(4,\cdot) = (2,1,1,0)$. To convert back, undo the shift. A path from u to v has, by the telescoping identity Equation (16.12) below, w'-weight equal to its w-weight plus $h(u) - h(v)$, so

$$\delta(u, v) \;=\; \hat{d}(u, v) - h(u) + h(v).$$

For $u = 4$ (with $h(4) = 0$): $\delta(4,1) = 2 - 0 + 0 = 2$, $\delta(4,2) = 1 - 0 + 0 = 1$, $\delta(4,3) = 1 - 0 + (-2) = -1$, $\delta(4,4) = 0$. The other three sources go the same way; the program below prints all four rows.

Solution to (d). Let $p = v_0 \to v_1 \to \cdots \to v_k$ be any path. Its reweighted weight is

$$w'(p) \;=\; \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \;=\; w(p) + h(v_0) - h(v_k), \qquad (16.12)$$

because every interior $h(v_i)$ is added once and subtracted once. The correction $h(v_0) - h(v_k)$ depends only on the two endpoints, so **every** path from v_0 to v_k is shifted by the *same* amount, and the ordering of those paths by weight — in particular, which one is shortest — is unchanged. Check it on the path $1 \to 2 \to 3$: $w = 4 - 2 = 2$, $w' = 4 + 0 = 4$, and indeed $2 + h(1) - h(3) = 2 + 0 + 2 = 4$.

Program. **Listing 16.14** performs all four steps.

Listing 16.14 Johnson's algorithm on the graph of Problem 16.12: Bellman–Ford from the auxiliary vertex, the reweighting with its arithmetic, and Dijkstra from every source, with the distances $\hat{d}$ under w' and the recovered distances D side by side.

```
// Problem 16.12: Johnson's algorithm on the four-vertex graph of Problem 16.11.
//   1. add a vertex q = 0 with an edge of weight 0 to every vertex
//   2. Bellman-Ford from q gives the potentials h(v) = delta(q, v)
//   3. reweight: w'(u,v) = w(u,v) + h(u) - h(v), which is never negative
//   4. Dijkstra from every vertex on w', then undo the shift
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
using namespace std;

const int INF = 1000000;
const int n = 4;
struct Edge { int u, v, w; };

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}};

    // Steps 1 and 2: Bellman-Ford from q over E plus the edges q->v of weight 0.
    vector<Edge> Eq;
    for (int v = 1; v <= n; ++v) Eq.push_back({0, v, 0});
    for (auto& e : E) Eq.push_back(e);
    vector<int> h(n + 1, INF);
    h[0] = 0;
    for (int pass = 1; pass <= n; ++pass) {          // |V'| - 1 = n passes
        for (auto& e : Eq)
            if (h[e.u] < INF && h[e.u] + e.w < h[e.v]) h[e.v] = h[e.u] + e.w;
        cout << "Bellman-Ford pass " << pass << ":";
        for (int v = 1; v <= n; ++v) cout << "  h(" << v << ")=" << setw(2) << h[v];
        cout << "\n";
    }
    for (auto& e : Eq)
        if (h[e.u] + e.w < h[e.v]) { cout << "negative cycle\n"; return 0; }

    // Step 3: reweight.
    cout << "\nreweighted edges:\n";
    vector<vector<pair<int,int>>> adj(n + 1);
    for (auto& e : E) {
        int w2 = e.w + h[e.u] - h[e.v];
        cout << "  w'(" << e.u << "," << e.v << ") = " << setw(2) << e.w << " + (" <<
setw(2)
             << h[e.u] << ") - (" << setw(2) << h[e.v] << ") = " << w2 << "\n";
        adj[e.u].push_back({e.v, w2});
    }
```

Listing 16.14 (Continued) *Johnson's algorithm on the graph of Problem 16.12: Bellman–Ford from the auxiliary vertex, the reweighting with its arithmetic, and Dijkstra from every source, with the distances \hat{d} under w' and the recovered distances D side by side.*

```
    // Step 4: Dijkstra from every source on w', then D[u][v] = dhat + h(v) - h(u).
    cout << "\n  source   dhat(u,1..4)          D(u,1..4)\n";
    for (int s = 1; s <= n; ++s) {
        vector<int> d(n + 1, INF);
        d[s] = 0;
        priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
        Q.push({0, s});
        while (!Q.empty()) {
            auto [du, u] = Q.top(); Q.pop();
            if (du > d[u]) continue;
            for (auto [v, w] : adj[u])
                if (d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
        }
        cout << setw(8) << s << "  ";
        for (int v = 1; v <= n; ++v) cout << setw(5) << d[v];
        cout << "     ";
        for (int v = 1; v <= n; ++v) cout << setw(5) << d[v] + h[v] - h[s];
        cout << "\n";
    }
    return 0;
}
```

It prints:

```
Bellman-Ford pass 1:  h(1)= 0  h(2)= 0  h(3)=-2  h(4)= 0
Bellman-Ford pass 2:  h(1)= 0  h(2)= 0  h(3)=-2  h(4)= 0
Bellman-Ford pass 3:  h(1)= 0  h(2)= 0  h(3)=-2  h(4)= 0
Bellman-Ford pass 4:  h(1)= 0  h(2)= 0  h(3)=-2  h(4)= 0

reweighted edges:
  w'(1,2) =  4 + ( 0) - ( 0) = 4
  w'(1,4) =  5 + ( 0) - ( 0) = 5
  w'(2,3) = -2 + ( 0) - (-2) = 0
  w'(3,1) =  3 + (-2) - ( 0) = 1
  w'(3,4) =  2 + (-2) - ( 0) = 0
  w'(4,2) =  1 + ( 0) - ( 0) = 1
  w'(4,3) =  6 + ( 0) - (-2) = 8

  source   dhat(u,1..4)          D(u,1..4)
       1      0    4    4    4         0    4    2    4
       2      1    0    0    0         1    0   -2    0
       3      1    1    0    0         3    3    0    2
       4      2    1    1    0         2    1   -1    0
```

The D columns are exactly the rows of $D^{(4)}$ from Problem 16.11. Two algorithms built on entirely different ideas — a dynamic program over allowed interior vertices, and a reduction to Dijkstra through potentials — agree on all sixteen entries.

Which would you use? On four vertices, it does not matter. In general, Floyd–Warshall costs $\Theta(V^3)$ and Johnson costs one Bellman–Ford, $O(VE)$, plus V runs of Dijkstra, $O(VE\lg V)$ with a binary heap. For a dense graph, $E = \Theta(V^2)$, Johnson's $O(V^3\lg V)$ loses to Floyd–Warshall's simple triple loop; for a sparse graph, $E = \Theta(V)$, Johnson's $O(V^2\lg V)$ wins by a factor of nearly V. Problem 16.17 in the next section compares the two in words and on a larger graph.

Check yourself. Change the weight of $1 \to 4$ from 5 to -1 (it creates no negative cycle). Find the new potentials, the reweighted edges, and the new first row of the distance matrix.

Answer. Bellman–Ford from q gives $h = (0,0,-2,-1)$: vertex 4 is now reached for -1 through $1 \to 4$. The reweighted edges are $w'(1,2) = 4$, $w'(1,4) = -1 + 0 + 1 = 0$, $w'(2,3) = 0$, $w'(3,1) = 1$,

$w'(3,4) = 2 - 2 + 1 = 1$, $w'(4,2) = 1 - 1 - 0 = 0$, $w'(4,3) = 6 - 1 + 2 = 7$ — all non-negative. Dijkstra from 1 gives $\hat{d}(1,\cdot) = (0,0,0,0)$, and converting back, $\delta(1,\cdot) = (0,\ 0 - 0 + 0,\ 0 - 0 - 2,\ 0 - 0 - 1) = (0,0,-2,-1)$. The route to 2 is now $1 \to 4 \to 2$ of weight $-1 + 1 = 0$.

16.11 Further Worked Problems

The nine problems of this section revisit the rest of the book in the same shapes as Problems 16.1 to 16.5: fifteen quick statements, an algorithm to design and analyze, an induction, a three-part proof about an algorithm's guarantee, and a comparison of two algorithms for the same task. They come in two groups, Problems 16.13 to 16.17 and Problems 16.18 to 16.21, and each group is worth attempting in one sitting, with the solutions covered, before reading on.

16.11.1 Problem 16.13 — fifteen more statements

Problem 16.13. Mark each statement **T** or **F**, with a one-line justification.

1. Kruskal's algorithm can be implemented to run in $O(E\lg V)$ time.
2. In a directed graph with a cycle, a topological sort still exists if the cycle has length 2.
3. A greedy algorithm always produces an optimal solution for the fractional knapsack problem.
4. A greedy algorithm always produces an optimal solution for the 0/1 knapsack problem.
5. Huffman coding assigns the shortest codeword to the most frequent symbol.
6. Breadth-first search computes shortest paths in an unweighted graph.
7. Depth-first search on an undirected graph can produce a cross edge.
8. Floyd–Warshall runs in $\Theta(V^3)$ time and uses $\Theta(V^2)$ space.
9. Johnson's algorithm is asymptotically better than Floyd–Warshall on a sparse graph.
10. The amortized cost of an operation can be larger than its worst-case cost.
11. In the potential method, the potential function may become negative provided it returns to zero at the end.
12. A dynamic table that doubles on overflow has $O(1)$ amortized insertion cost.
13. Relaxing an edge can increase $d[v]$.
14. Every edge of a minimum spanning tree is a lightest edge across some cut.
15. A shortest path in a graph with no negative cycles never repeats a vertex.

Solution. (1) **T** — sorting the E edges dominates, $O(E\lg E) = O(E\lg V)$ since $E < V^2$ so $\lg E < 2\lg V$. (2) **F** — any cycle, of any length, makes a topological sort impossible; a 2-cycle $u \to v \to u$ is a cycle. (3) **T** — take items in decreasing value-per-unit-weight and split the last one; the exchange argument proves it optimal. (4) **F** — with items $(w, v) = (10,60), (20,100), (30,120)$ and capacity 50, greedy by ratio takes the first two for 160, but $\{2,3\}$ gives 220. (5) **T** — a symbol of higher frequency is merged later, so it sits nearer the root; formally, if $f(a) > f(b)$ then $\text{depth}(a) \le \text{depth}(b)$ in the Huffman tree. (6) **T** — BFS computes fewest-edge paths, which are shortest when every weight is 1. (7) **F** — undirected DFS produces only tree edges and back edges; a would-be cross edge would have been explored from the other end first. (8) **T** — three nested loops, one $V \times V$ matrix if updated in place. (9) **T** — Johnson is $O(V^2\lg V + VE)$ with a Fibonacci heap; on a sparse graph with $E = \Theta(V)$ that is $O(V^2\lg V)$, against $\Theta(V^3)$. (10) **T** — the accounting method deliberately overcharges cheap operations; a stack PUSH has worst-case cost 1 and amortized cost 2. (11) **F** — the potential method requires $\Phi(D_i) \ge \Phi(D_0)$ for *every* i, not merely at the end, so that Equation (16.7) bounds the real total for every prefix of the sequence; Problem 16.10(f) shows what goes wrong otherwise. (12) **T** — n insertions cost less than $3n$

(Problem 16.10), so $O(1)$ each. (13) **F** — relaxation only ever lowers $d[v]$; it is guarded by $d[v] > d[u] + w(u, v)$. (14) **T** — that is the cut property read backward: for $e \in T$, deleting e splits T into two components, and e is a lightest edge across that cut. (15) **T** — repeating a vertex means including a cycle, whose weight is ≥ 0, so deleting it does not lengthen the path.

16.11.2 Problem 16.14 — the longest path in a DAG

Problem 16.14. Design a polynomial-time algorithm that finds the longest path from a given source vertex in a weighted **directed acyclic graph**. Give assumptions, pseudocode and a time-complexity analysis, and explain why the same approach fails on a general directed graph.

What it tests. Problem 16.8 turned inside out: the DAG algorithm works with *any* weights, so it works with the negated weights too, and "shortest with $-w$" is "longest with w".

Solution. *Assumptions.* The input is a directed acyclic graph $G = (V, E)$ with real edge weights w and a source s; "longest path" means maximum total weight; a path visits no vertex twice; vertices not reachable from s have length $-\infty$.

Design. A DAG has a topological order — a linear order in which every edge goes forward (Chapter 10). Process the vertices in that order and relax every out-edge, taking the **maximum** instead of the minimum, as in **Pseudocode 16.5**.

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)
 topologically sort V(G)
 for each v in V:  dist[v] = -infinity;  par[v] = NIL
 dist[s] = 0
 for each u in topological order
     if dist[u] > -infinity
         for each edge (u,v) in Adj[u]
             if dist[u] + w(u,v) > dist[v]
                 dist[v] = dist[u] + w(u,v);  par[v] = u
 return dist, par
```

Correctness. When u is processed, every predecessor of u has already been processed, because a topological order puts the tail of every edge before its head. So $dist[u]$ is final when it is used. Formally, by induction over the topological order: the claim *"$dist[u]$ equals the longest-path weight from s to u when u is processed"* holds for the first vertex and is maintained because every in-edge of u has been relaxed exactly once before u is reached.

Complexity. Topological sort is $\Theta(V + E)$; the relaxation loop touches each edge exactly once, so $\Theta(V + E)$ total. Space $\Theta(V)$.

Why it fails on a general digraph. Two reasons, and both should be given. First, there is no topological order, so the "process predecessors first" argument has nothing to stand on. Second and more deeply, **longest simple path in a general digraph is NP-hard**: a Hamiltonian path is a longest simple path with all weights 1. The acyclicity is not a convenience; it is the whole reason the problem is easy.

A worked instance. Take $V = \{1, \ldots, 7\}$ and edges $1 \to 2$ (6), $1 \to 3$ (4), $2 \to 4$ (5), $2 \to 5$ (9), $3 \to 4$ (3), $3 \to 5$ (7), $4 \to 6$ (2), $4 \to 7$ (8), $5 \to 6$ (4), $6 \to 7$ (3). In the topological order 1,3,2,5,4,6,7:

$$dist[3] = 4, \quad dist[2] = 6, \quad dist[5] = \max(6+9, 4+7) = 15, \quad dist[4] = \max(6+5, 4+3) = 11,$$

$$dist[6] = \max(11+2, 15+4) = 19, \quad dist[7] = \max(11+8, 19+3) = 22.$$

The longest path is $1 \to 2 \to 5 \to 6 \to 7$ with weight $6 + 9 + 4 + 3 = 22$. **Listing 16.15** confirms it, using Kahn's queue-based topological sort rather than the DFS of Problem 16.8, so that both methods of Chapter 10 appear in this chapter.

Listing 16.15 The longest path in a weighted DAG for Problem 16.14: Kahn's topological sort followed by one maximizing relaxation per edge.

```
// Problem 16.14: the longest path in a weighted DAG, by topological order.
// Topological sort (Ch. 11) + one relaxation per edge (Ch. 13) = Theta(V+E).
#include <iostream>
#include <iomanip>
#include <vector>
#include <stack>
#include <climits>
using namespace std;

const int NEG = INT_MIN / 4;

int main() {
    const int n = 7;                              // vertices 1..7
    vector<vector<pair<int,int>>> adj(n + 1);
    auto add = [&](int u, int v, int w) { adj[u].push_back({v, w}); };
    add(1,2,6); add(1,3,4); add(2,4,5); add(2,5,9); add(3,4,3);
    add(3,5,7); add(4,6,2); add(4,7,8); add(5,6,4); add(6,7,3);

    vector<int> indeg(n + 1, 0);
    for (int u = 1; u <= n; ++u) for (auto [v, w] : adj[u]) ++indeg[v];

    vector<int> order;                            // Kahn's topological sort
    vector<int> ind = indeg;
    stack<int> S;
    for (int v = 1; v <= n; ++v) if (ind[v] == 0) S.push(v);
    while (!S.empty()) {
        int u = S.top(); S.pop();
        order.push_back(u);
        for (auto [v, w] : adj[u]) if (--ind[v] == 0) S.push(v);
    }
    cout << "topological order:";
    for (int v : order) cout << " " << v;
    cout << "\n\n";

    vector<int> dist(n + 1, NEG), par(n + 1, 0);
    dist[1] = 0;
    cout << "relaxing in topological order\n  v  dist[v]  via\n";
    for (int u : order) {
        if (dist[u] == NEG) continue;             // unreachable from the source
        for (auto [v, w] : adj[u])
            if (dist[u] + w > dist[v]) { dist[v] = dist[u] + w; par[v] = u; }
    }
    for (int v : order)
        cout << setw(3) << v << setw(9) << dist[v] << setw(6) << par[v] << "\n";
```

Listing 16.15 (Continued) *The longest path in a weighted DAG for Problem 16.14: Kahn's topological sort followed by one maximizing relaxation per edge.*

```
    int best = 1;
    for (int v = 1; v <= n; ++v) if (dist[v] > dist[best]) best = v;
    vector<int> path;
    for (int v = best; v != 0; v = par[v]) path.push_back(v);
    cout << "\nlongest path has weight " << dist[best] << ":";
    for (int i = (int)path.size() - 1; i >= 0; --i) cout << " " << path[i];
    cout << "\n";
    return 0;
}
```

```
topological order: 1 3 2 5 4 6 7

relaxing in topological order
  v  dist[v]  via
  1        0    0
  3        4    1
  2        6    1
  5       15    2
  4       11    2
  6       19    5
  7       22    6

longest path has weight 22: 1 2 5 6 7
```

This is the critical-path method of project scheduling (Section 13.9.7): if the vertices are milestones and the weights are durations, the longest path is the shortest possible length of the whole project.

Check yourself. Add the edge $3 \to 6$ of weight 20. What is the longest path now?

Answer. $dist[6] = \max(11+2,\ 15+4,\ 4+20) = 24$ and $dist[7] = \max(11+8,\ 24+3) = 27$: the path $1 \to 3 \to 6 \to 7$ of weight $4+20+3 = 27$. Note that the topological order 1,3,2,5,4,6,7 is still valid, since the new edge goes forward.

16.11.3 Problem 16.15 — a tree with n vertices has $n-1$ edges

Problem 16.15. Prove by induction that a tree — a connected acyclic undirected graph — with n vertices has exactly $n-1$ edges.

Solution. *Claim.* Every connected acyclic undirected graph on $n \geq 1$ vertices has exactly $n-1$ edges. *Proof, by strong induction on n.*

Base, $n = 1$. A single vertex, connected and acyclic, has no edges, and $1 - 1 = 0$.

Hypothesis. Assume every tree on n' vertices has $n'-1$ edges, for all $1 \leq n' < n$.

Step. Let T be a tree on $n \geq 2$ vertices. First, T has a vertex of degree 1. (Walk from any vertex without immediately reversing; because T is finite and acyclic the walk cannot return to a visited vertex, so it must stop, and it can only stop at a vertex with no unused edge — a leaf.) Let ℓ be such a vertex and e its single edge. Remove ℓ and e to get T'.

T' is acyclic, since removing things cannot create a cycle. T' is connected: any path in T between two vertices of T' cannot pass *through* ℓ, because entering ℓ uses its only edge and leaving would have to reuse it. And T' has $n-1$ vertices, with $1 \leq n-1 < n$, so the hypothesis gives $|E(T')| = (n-1)-1 = n-2$. Adding back ℓ and e,

$$|E(T)| = |E(T')| + 1 = n - 2 + 1 = n - 1. \qquad \blacksquare$$

The alternative, in two lines. Build T by adding vertices one at a time, each joined to the existing part by exactly one edge — one edge per vertex after the first. This is the same induction stated forward. And it is the fact used silently in Problem 16.4: deleting one edge from a tour of n vertices leaves $n-1$ edges on n vertices, connected — a spanning tree.

16.11.4 Problem 16.16 — Huffman coding, in three parts

Problem 16.16. Huffman's algorithm repeatedly merges the two least frequent symbols. Prove (1) that the code produced is prefix-free, so it can be decoded unambiguously; (2) that the running time is $O(n\lg n)$ for n symbols; (3) that the code produced has minimum expected length among all prefix-free codes.

Solution (1): prefix-free. In the tree Huffman builds, a symbol is placed only at a **leaf**: every merge creates a new *internal* node whose children are the two trees merged, and symbols are never moved off leaves. A codeword is the sequence of left/right choices on the root-to-leaf path. If codeword x were a prefix of codeword y, then the leaf for x would lie on the path to the leaf for y — impossible, because a leaf has no descendants. Hence no codeword is a prefix of another, and decoding is unambiguous: read bits until a leaf is reached, emit its symbol, restart at the root. ■

Solution (2): $O(n\lg n)$. Build a min-priority queue of the n symbols: $O(n)$ with `Build-Min-Heap`. The loop runs $n-1$ times; each iteration does two `Extract-Min` and one `Insert`, each $O(\lg n)$. Total $O(n)+O(n\lg n)=O(n\lg n)$. If the frequencies arrive already sorted, two ordinary queues replace the heap and the whole algorithm is $\Theta(n)$.

Solution (3): optimality. Two lemmas, then an induction — the standard shape of every greedy proof (Chapter 9). Write $f(c)$ for the frequency of symbol c and $d(c)$ for its depth in the tree; the cost of a tree is $\sum_c f(c)\,d(c)$.

Lemma A (greedy choice). Let x and y be the two symbols of lowest frequency. There is an optimal prefix code in which x and y are siblings at maximum depth. *Proof.* Take any optimal tree T and let a, b be two siblings at maximum depth (a deepest internal node has two leaf children, since an optimal tree is full). Exchange x with a. The cost changes by

$$\begin{aligned} f(x)\,d(a)+f(a)\,d(x)-f(x)\,d(x)-f(a)\,d(a) \\ = \; -(f(a)-f(x))(d(a)-d(x)) \;\le\; 0, \end{aligned}$$

because $f(x)\le f(a)$ and $d(x)\le d(a)$. So the exchange does not increase the cost. Do the same for y and b. The result is still optimal and has x, y as deepest siblings. ▫

Lemma B (optimal substructure). Let T be an optimal tree for the alphabet C, and let x, y be siblings in it with parent z. Let $C' = C \setminus \{x,y\} \cup \{z\}$ with $f(z)=f(x)+f(y)$, and let T' be T with the leaves x, y deleted. Then $\text{cost}(T)=\text{cost}(T')+f(x)+f(y)$, and T' is optimal for C'. *Proof.* The cost identity is arithmetic: x and y sit one level below z, so $f(x)d(x)+f(y)d(y)=(f(x)+f(y))(d(z)+1)=f(z)d(z)+f(x)+f(y)$. For optimality, if some T'' beat T' on C', re-attaching x and y under z in T'' would beat T on C. ▫

Induction. On $n=|C|$. For $n=2$ any tree assigns one bit to each symbol and all are optimal. For $n>2$, Huffman merges x, y into z and recurses on C', which has $n-1$ symbols; by induction it returns an optimal tree T' for C'; by Lemma B the tree obtained by re-attaching x and y costs $\text{cost}(T')+f(x)+f(y)$, and by Lemma A some optimal tree for C has x, y as deepest siblings and hence, again by Lemma B, costs at least that. So Huffman's tree is optimal. ■

Exercise 8 of Section 16.19 builds a Huffman code for six Profitina Laundry service tags and checks the arithmetic with a program.

16.11.5 Problem 16.17 — Floyd–Warshall against Johnson

Problem 16.17. Floyd–Warshall and Johnson both solve the all-pairs shortest-paths problem. (1) Describe their differences and when each should be chosen; (2) write pseudocode for each and show where they differ; (3) illustrate each on a small graph.

Solution (1): differences. Floyd–Warshall is a dynamic program over *which vertices may be interior to a path*, Equation (16.10). It is three nested loops, $\Theta(V^3)$ time and $\Theta(V^2)$ space, it accepts negative edges, and it detects a negative cycle by a negative diagonal entry. Johnson is a *reduction*: run Bellman–Ford once from an added vertex to get a potential h, reweight every edge by Equation (16.11) — which makes every weight non-negative and preserves shortest paths — then run Dijkstra from each vertex. That is $O(VE)$ for the Bellman–Ford plus V Dijkstra runs, giving $O(V^2 \lg V + VE)$ with a Fibonacci heap. **Choose Floyd–Warshall when the graph is dense or V is small; choose Johnson when the graph is sparse.** At $V = 1000$, $E = 3000$: Floyd–Warshall does 10^9 basic steps, Johnson about 10^7.

Solution (2): pseudocode. **Pseudocode 16.6** sets the two side by side.

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
```

The difference to point at: Floyd–Warshall never changes the weights and pays $\Theta(V^3)$; Johnson changes the weights once so that Dijkstra becomes legal, and pays V Dijkstra runs. The reweighting is sound because the potentials telescope along any path, Equation (16.12), so every path between the same two endpoints shifts by the same constant and the minimizer is unchanged. Non-negativity of w' follows from the triangle inequality on h: $h(v) \le h(u) + w(u, v)$.

Solution (3): illustration. Problems 16.11 and 16.12 have already run both algorithms by hand on a four-vertex graph. **Listing 16.16** runs both on a larger, five-vertex graph with two negative edges, printing $D^{(0)}$ through $D^{(5)}$, the potentials, the reweighted edges, and Johnson's answer.

Listing 16.16 Floyd–Warshall and Johnson on one five-vertex graph, with all six intermediate matrices, the potentials and the reweighted edges printed.

```
// Problem 16.17: Floyd-Warshall against Johnson on the same graph.
// Five vertices, one negative edge, no negative cycle.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

const int INF = 1000000;
const int n = 5;
struct Edge { int u, v, w; };
vector<Edge> E = {{1,2,3},{1,3,8},{1,5,-4},{2,4,1},{2,5,7},
                  {3,2,4},{4,1,2},{4,3,-5},{5,4,6}};

void printMat(const string& tag, int D[6][6]) {
    cout << tag << "\n";
    for (int i = 1; i <= n; ++i) {
        cout << "    ";
        for (int j = 1; j <= n; ++j) {
            if (D[i][j] >= INF) cout << setw(5) << "inf";
            else                cout << setw(5) << D[i][j];
        }
        cout << "\n";
    }
    cout << "\n";
}

int main() {
    int D[6][6];
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j) D[i][j] = (i == j) ? 0 : INF;
    for (auto& e : E) D[e.u][e.v] = min(D[e.u][e.v], e.w);
    printMat("D(0)  -- the weight matrix W", D);

    for (int k = 1; k <= n; ++k) {
        for (int i = 1; i <= n; ++i)
            for (int j = 1; 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];   // guard: inf + w is still inf
        printMat("D(" + to_string(k) + ")", D);
    }
```

Listing 16.16 (Continued) *Floyd–Warshall and Johnson on one five-vertex graph, with all six intermediate matrices, the potentials and the reweighted edges printed.*

```
    // ---- Johnson: reweight with h from Bellman-Ford, then run Dijkstra n times.
    vector<int> h(n + 1, 0);                    // h = distances from a new vertex 0
    for (int pass = 1; pass <= n; ++pass)       // edges 0->v all have weight 0
        for (auto& e : E)
            if (h[e.u] + e.w < h[e.v]) h[e.v] = h[e.u] + e.w;
    cout << "Johnson's potentials h(v):";
    for (int v = 1; v <= n; ++v) cout << "  h(" << v << ")=" << h[v];
    cout << "\n\nreweighted edges  w'(u,v) = w(u,v) + h(u) - h(v):\n   ";
    for (auto& e : E)
        cout << e.u << "->" << e.v << "=" << e.w + h[e.u] - h[e.v] << "  ";
    cout << "\n\n";

    vector<vector<pair<int,int>>> adj(n + 1);
    for (auto& e : E) adj[e.u].push_back({e.v, e.w + h[e.u] - h[e.v]});

    int J[6][6];
    for (int s = 1; s <= n; ++s) {
        vector<int> d(n + 1, INF); d[s] = 0;
        priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
        Q.push({0, s});
        while (!Q.empty()) {
            auto [du, u] = Q.top(); Q.pop();
            if (du > d[u]) continue;
            for (auto [v, w] : adj[u])
                if (d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
        }
        for (int v = 1; v <= n; ++v)
            J[s][v] = (d[v] >= INF) ? INF : d[v] - h[s] + h[v];   // undo the shift
    }
    printMat("Johnson's answer", J);

    bool same = true;
    for (int i = 1; i <= n; ++i) for (int j = 1; j <= n; ++j) if (D[i][j] != J[i][j]) same
= false;
    cout << "Floyd-Warshall and Johnson agree on every entry: "
         << (same ? "yes" : "no") << "\n";
    return 0;
}
```

Its output:

```
D(0)  -- the weight 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(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

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

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

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

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

Johnson's potentials h(v):  h(1)=0  h(2)=-1  h(3)=-5  h(4)=0  h(5)=-4

reweighted edges  w'(u,v) = w(u,v) + h(u) - h(v):
   1->2=4  1->3=13  1->5=0  2->4=0  2->5=10  3->2=0  4->1=2  4->3=0  5->4=2

Johnson's answer
       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

Floyd-Warshall and Johnson agree on every entry: yes
```

Note that every reweighted edge is non-negative — 4,13,0,0,10,0,2,0,2 — which is exactly the condition Dijkstra needs, manufactured on purpose.

16.11.6 Problem 16.18 — fifteen statements again

Problem 16.18. Mark each statement **T** or **F**, with a one-line justification.

1. Heapsort sorts in place and runs in $\Theta(n\lg n)$ in every case.
2. `BUILD-MAX-HEAP` runs in $\Theta(n\lg n)$.

3. An AVL tree of height h contains at least $F(h+3)-1$ nodes, where F is the Fibonacci sequence.
4. The inorder walk of a binary search tree prints the keys in sorted order.
5. Any comparison sort makes $\Omega(n\lg n)$ comparisons in the worst case.
6. Memoization and bottom-up tabulation always have the same asymptotic running time.
7. A problem with optimal substructure can always be solved greedily.
8. A disjoint-set forest with union by rank and path compression supports m operations on n elements in $O(m\,\alpha(n))$.
9. In Prim's algorithm the set of chosen edges always forms a single tree.
10. In Kruskal's algorithm the set of chosen edges always forms a single tree.
11. Dijkstra's algorithm visits every edge exactly once.
12. The aggregate, accounting and potential methods can give different amortized costs for the same operation.
13. The 0/1 knapsack dynamic program runs in polynomial time in the size of its input.
14. A graph with V vertices and more than $V-1$ edges must contain a cycle.
15. Removing the heaviest edge of a cycle never disconnects a connected graph.

Solution. (1) **T**. (2) **F** — it is $\Theta(n)$; most nodes are near the bottom, and $\sum_h \lceil n/2^{h+1}\rceil\, O(h) = O(n)$. (3) **T** — from $N(h) = 1 + N(h-1) + N(h-2)$. (4) **T**. (5) **T** — the decision tree has $n!$ leaves, so its height is $\geq \lg(n!) = \Theta(n\lg n)$. (6) **F** — they agree when every subproblem is needed, but memoization may skip subproblems the table would fill, and can therefore be asymptotically faster on some inputs; tabulation avoids recursion overhead and can be faster in practice on others. (7) **F** — optimal substructure is necessary but not sufficient; the greedy-choice property is the extra hypothesis, and 0/1 knapsack has the first without the second. (8) **T**. (9) **T** — Prim grows one tree from the root; the chosen edges are connected at every step. (10) **F** — Kruskal maintains a *forest* that only becomes a single tree at the very last edge. (11) **F** — it relaxes every edge at most once, but edges from unreachable vertices are never relaxed at all. (12) **T** — the three methods may charge the *same* operation differently: the aggregate method gives every operation the same average cost, whereas the accounting and potential methods can charge `PUSH` 2 and `POP` 0. What they must agree on is that the amortized total bounds the true total, not how the total is shared out; in Problem 16.9 they happened to agree call by call. (13) **F** — $\Theta(nW)$ is *pseudo*-polynomial: W is exponential in the number of bits used to write it, so the running time is not polynomial in the input size. Problem 16.19 is about exactly this. (14) **T** — a forest on V vertices has at most $V-1$ edges (Problem 16.15, applied to each component), so more than $V-1$ edges forces a cycle. (15) **T** — the cycle property: the two endpoints stay connected around the rest of the cycle.

16.11.7 Problem 16.19 — the 0/1 knapsack, and what "polynomial" means

Problem 16.19. Design an algorithm for the 0/1 knapsack problem: n items with integer weights w_i and values v_i, and an integer capacity W; choose a subset of maximum total value with total weight at most W. Give assumptions, pseudocode, a time-complexity analysis, and a carefully justified statement of whether your algorithm runs in polynomial time.

Solution. *Assumptions.* w_i, v_i, W are non-negative integers; each item may be taken at most once (that is what 0/1 means); the value of the empty set is 0.

Design. Let $c[i][j]$ be the best value obtainable from the first i items with capacity j available. Item i is either left out — value $c[i-1][j]$ — or taken, which needs $w_i \le j$ and yields $v_i + c[i-1][j - w_i]$:

$$c[i][j] = \begin{cases} 0 & i = 0 \text{ or } j = 0, \\ c[i-1][j] & w_i > j, \\ \max(c[i-1][j],\ v_i + c[i-1][j-w_i]) & \text{otherwise.} \end{cases} \tag{16.13}$$

Pseudocode. **Pseudocode 16.7** fills the table of Equation (16.13) row by row.

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)
for j = 0 to W:  c[0][j] = 0
for i = 1 to n
  c[i][0] = 0
  for j = 1 to W
    if w[i] > j: c[i][j] = c[i-1][j]
    else:        c[i][j] = max(c[i-1][j], v[i] + c[i-1][j-w[i]])
return c[n][W]
```

Complexity. Two nested loops of n and W iterations with a $\Theta(1)$ body: $\Theta(nW)$ time and $\Theta(nW)$ space, reducible to $\Theta(W)$ space by keeping one row.

Is it polynomial? **No, and saying yes is the trap.** The input consists of $2n + 1$ numbers. Writing W takes $\lceil \lg(W + 1) \rceil$ bits, so if the input has b bits then W can be as large as 2^b. A running time of $\Theta(nW)$ is therefore exponential in the input *size*, even though it is polynomial in the input *value*. Such an algorithm is called **pseudo-polynomial**. It is fast when W is small — a 60-minute machine schedule, a 100-rupiah coin system — and useless when W is a 64-bit number. 0/1 knapsack is NP-hard, and a genuinely polynomial algorithm would prove P = NP.

Worked instance. Five Profitina Laundry add-on services, each consuming machine-minutes and returning value in thousands of rupiah, with a ten-minute budget: starch (3,30), express (4,50), perfume (5,60), fold (2,20), iron (6,70). **Listing 16.17** fills the table.

Listing 16.17 The 0/1 knapsack dynamic program of Problem 16.19, printing the whole c table and reconstructing the chosen set by walking it backward.

```
// 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
```

Check by hand: express and iron give $50 + 70 = 120$ at $4 + 6 = 10$ minutes, and no other subset fitting in ten minutes beats it — { perfume, express } gives 110 in 9 minutes, { perfume, starch, fold } gives 110 in 10. Note also the reconstruction rule: $c[i][j] \neq c[i-1][j]$ means item i was taken.

Check yourself. Raise the budget to 11 minutes. What is the best value, and which services?

Answer. 130, reached by two different subsets: perfume and iron ($60 + 70$ at $5 + 6 = 11$ minutes), and express, perfume and fold ($50 + 60 + 20$ at $4 + 5 + 2 = 11$ minutes). The table only computes $c[5][11] = 130$; which of the two optimal sets the backward walk reports depends on the order of the items, just as the longest increasing subsequence of Problem 16.2 was not unique.

16.11.8 Problem 16.20 — a 2-approximation for vertex cover

Problem 16.20. `APPROX-VERTEX-COVER` repeatedly picks an arbitrary uncovered edge (u, v), adds **both** u and v to the cover, and deletes every edge incident to u or v. Prove (1) that the set returned is a vertex cover; (2) that the running time is polynomial; (3) that the algorithm has ratio bound $\rho(n) = 2$.

Solution (1): it returns a vertex cover. The loop ends only when no edge is left uncovered — the algorithm deletes an edge from consideration exactly when one of its endpoints has entered C. So at termination every edge of the original graph has at least one endpoint in C, which is the definition of a vertex cover. ■

Solution (2): polynomial. With adjacency lists, each edge is examined a constant number of times: once when picked, and once when deleted because one of its endpoints was chosen. So the running time is $O(V + E)$. ■

Solution (3): ratio bound 2. Let A be the set of edges the algorithm *picked* (not the ones it deleted). Two observations.

A is a matching. When edge (u, v) is picked, every edge touching u or v is deleted, so no later picked edge shares an endpoint with it. Hence the edges of A are pairwise disjoint.

Every vertex cover contains at least one endpoint of each edge of A, and these are distinct. Let C^* be an optimal cover. Each edge of A must be covered, so C^* contains at least one of its two endpoints; and since the edges of A share no endpoints, these are $|A|$ distinct vertices. Therefore $|C^*| \geq |A|$.

The algorithm returns $2|A|$ vertices, two per picked edge, all distinct by the matching property. So

$$|C| = 2|A| \leq 2|C^*|, \qquad \frac{|C|}{|C^*|} \leq 2. \qquad \blacksquare$$

The proof has the same skeleton as Problem 16.4: find a quantity that lower-bounds the optimum ($|A|$ here, $d(T)$ there), and show the algorithm's answer is at most twice it. **Listing 16.18** runs it against brute force.

Listing 16.18 The 2-approximation for vertex cover of Problem 16.20, with the matching it implicitly builds and the true optimum found by exhaustive search.

```
// Problem 16.20: APPROX-VERTEX-COVER. Take any uncovered edge, put BOTH
// endpoints in the cover, discard every edge they touch, repeat. Ratio bound 2.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <set>
using namespace std;

struct Edge { int u, v; };

int main() {
    const int n = 7;
    string nm[n] = {"a","b","c","d","e","f","g"};
    vector<Edge> E = {{0,1},{1,2},{2,3},{3,4},{3,5},{4,5},{4,6},{2,6}};

    // ---- the approximation
    set<int> C;
    vector<bool> covered(E.size(), false);
    vector<Edge> matching;
    for (size_t i = 0; i < E.size(); ++i) {
        if (covered[i]) continue;
        int u = E[i].u, v = E[i].v;
        C.insert(u); C.insert(v);
        matching.push_back(E[i]);
        cout << "pick edge (" << nm[u] << "," << nm[v] << ")  ->  cover gains "
             << nm[u] << " and " << nm[v] << "\n";
        for (size_t k = 0; k < E.size(); ++k)
            if (E[k].u == u || E[k].v == u || E[k].u == v || E[k].v == v) covered[k] =
true;
    }
    cout << "\napproximate cover C =";
    for (int v : C) cout << " " << nm[v];
    cout << "   |C| = " << C.size()
         << "   (matching size " << matching.size() << ")\n";

    // ---- the optimum, by brute force over all 2^n subsets
    int best = n + 1, bestMask = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        bool ok = true;
        for (auto& e : E)
            if (!((mask >> e.u) & 1) && !((mask >> e.v) & 1)) { ok = false; break; }
        if (!ok) continue;
        int sz = __builtin_popcount(mask);
        if (sz < best) { best = sz; bestMask = mask; }
    }
    cout << "optimal cover C* =";
```

Listing 16.18 (Continued) *The 2-approximation for vertex cover of Problem 16.20, with the matching it implicitly builds and the true optimum found by exhaustive search.*

```
    for (int v = 0; v < n; ++v) if ((bestMask >> v) & 1) cout << " " << nm[v];
    cout << "   |C*| = " << best << "\n";

    cout << fixed << setprecision(4)
         << "\nratio |C|/|C*| = " << (double)C.size() / best
         << "   (the theorem guarantees at most 2)\n"
         << "check |C*| >= matching size:  " << best << " >= " << matching.size() << "\n"
         << "check |C| = 2 * matching:     " << C.size() << " = " << 2 * matching.size() <<
"\n";
    return 0;
}
```

```
pick edge (a,b)  ->  cover gains a and b
pick edge (c,d)  ->  cover gains c and d
pick edge (e,f)  ->  cover gains e and f

approximate cover C = a b c d e f   |C| = 6   (matching size 3)
optimal cover C* = a c d e   |C*| = 4

ratio |C|/|C*| = 1.5000   (the theorem guarantees at most 2)
check |C*| >= matching size:  4 >= 3
check |C| = 2 * matching:     6 = 6
```

16.11.9 Problem 16.21 — Prim against Kruskal

Problem 16.21. Prim's and Kruskal's algorithms both build a minimum spanning tree. (1) Describe their differences and when each should be chosen; (2) write pseudocode for each and show where they differ; (3) illustrate each on a small graph.

Solution (1): differences. Both are instances of the generic MST algorithm of Chapter 11: repeatedly add a **safe** edge, justified by the cut property. Prim grows a *single tree* from a root, always taking the lightest edge leaving it; it needs a priority queue and runs in $\Theta(V^2)$ with an array or $O(E\lg V)$ with a binary heap. Kruskal grows a *forest*, sorting all edges and taking each if it joins two different components; it needs a disjoint-set forest and runs in $O(E\lg E) = O(E\lg V)$, dominated by the sort. **Choose Prim on dense graphs**, where $\Theta(V^2)$ beats $O(V^2\lg V)$ and no sort is needed; **choose Kruskal on sparse graphs**, or when the edges arrive already sorted, in which case Kruskal is $O(E\,\alpha(V))$ — effectively linear, by the amortized analysis of union–find in Section 14.8.

Solution (2): pseudocode. **Pseudocode 16.8** sets them side by side.

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
```

The structural difference is in line 5 of each. Prim's `EXTRACT-MIN` asks *"which vertex is nearest to the tree I already have?"*; Kruskal's `FIND-SET` asks *"does this edge join two things that are not yet joined?"* Prim is local and incremental; Kruskal is global and sorted. Prim's line 8 is also a relative of relaxation: Dijkstra would set `key[v] = key[u] + w(u,v)`, Prim sets `key[v] = w(u,v)` — distance to the tree rather than distance to the source.

Solution (3): illustration. Use the six laundry points of **Table 16.5**. Both algorithms produce the same tree of weight 22 — and because that tree is unique here, they must. **Table 16.17** puts their traces side by side.

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

The instructive detail: three edges have weight 5, so the uniqueness theorem of Section 16.3.9 does not apply — and yet an exhaustive search over all $\binom{15}{5} = 3003$ five-edge subsets finds exactly one spanning tree of weight 22. Ties permit a second minimum spanning tree; they do not produce one.

16.12 The Problems of This Chapter Compared

Having worked twenty-one problems, step back and ask what each one used. **Table 16.18** lays them side by side.

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)	$\Theta(VE)$; $O(E\lg V)$
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)	$\Theta(VE)$
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
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)	$O(V^2\lg V + VE)$
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)	$\Theta(V^3)$; $O(V^2\lg V + VE)$
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 is common to all of them.

1. **Every one has an obvious approach that is wrong or too weak.** Answering statement (11) by the name of the notation, solving the contiguous subsequence problem, inducting on height, asserting that shortcutting shortens, trusting Dijkstra with a negative edge, bounding the counter by nk, choosing the potential $num - size$, adding a constant to every edge instead of a potential. Much of the value of each solution lies in *saying why* the obvious approach fails.
2. **Every answer was checked twice.** A proof and a program; a hand trace and a printed table; Floyd–Warshall against Johnson; the approximation against brute force; three amortized methods against one another.
3. **The same few ideas recur.** Relaxation appears in seven problems (16.5 to 16.8, 16.12, 16.14, 16.17); a strengthened hypothesis (a subproblem that *ends at* $S[i]$, a counter total *plus* its 1 bits) in two; a lower bound on the optimum in two; a potential function — Johnson's h and Chapter 14's Φ are the same idea wearing two coats — in three.

> **ONE IDEA, TWO CHAPTERS.** Johnson's potential h and the amortized potential Φ do the same job. Each adds to every step a correction that *telescopes* along any sequence of steps — $h(u) - h(v)$ along a path, $\Phi(D_i) - \Phi(D_{i-1})$ along a sequence of operations — so that the total changes only by a boundary term. Compare Equation (16.12) with Equation (16.7): in both, the interior terms cancel and only the two ends survive. Johnson uses the correction to make every edge non-negative; the potential method uses it to make every operation cheap. Once you see that they are the same trick, you will find it again in the dual of a linear program (Section 16.15).

16.13 Quick Self-Check Drills

Forty questions with one-line answers, spanning all sixteen chapters. Cover the answers, work down the list, and check yourself. Anything you miss names the chapter to reread. Two or three drills per chapter, in chapter order.

1. **The five phases of the problem-solving framework?** Understand, abstract, design, analyze, implement. (Ch. 1)
2. **Why read the constraints before designing?** They are a time budget: $n \le 10^6$ demands $O(n)$ or $O(n\lg n)$; $n \le 500$ permits $O(n^3)$. (Ch. 1, 4)
3. **What does the RAM model charge for an array subscript?** One unit, like any arithmetic operation or comparison. (Ch. 2)
4. **The three obligations of a loop invariant?** Initialization, maintenance, termination. (Ch. 2)
5. **Best and worst case of insertion sort, and the input causing each?** $\Theta(n)$ on a sorted array, $\Theta(n^2)$ on a reverse-sorted one. (Ch. 2)
6. **Define $f(n) = O(g(n))$.** There exist $c > 0, n_0$ with $0 \le f(n) \le c\,g(n)$ for all $n \ge n_0$. (Ch. 3)
7. **O against Ω in one sentence?** O bounds from above, Ω from below, Θ from both. (Ch. 3, Section 16.3.10)
8. **The three Master Theorem cases?** $f = O(n^{\log_b a - \varepsilon}) \Rightarrow \Theta(n^{\log_b a})$; $f = \Theta(n^{\log_b a}) \Rightarrow \Theta(n^{\log_b a}\lg n)$; $f = \Omega(n^{\log_b a + \varepsilon})$ with regularity $\Rightarrow \Theta(f)$. (Ch. 3)
9. **Solve $T(n) = 2T(n/2) + \Theta(n)$ and $T(n) = T(n-1) + \Theta(n)$.** $\Theta(n\lg n)$ and $\Theta(n^2)$ — merge sort and worst-case quicksort. (Ch. 3, 5)
10. **How should a program be checked before it is trusted?** Against the sample, then against the smallest and largest legal inputs, then against a brute force on random small cases. (Ch. 4)
11. **Why is `BUILD-MAX-HEAP` $\Theta(n)$?** Most nodes are near the bottom; $\sum_h \lceil n/2^{h+1} \rceil O(h) = O(n)$. (Ch. 5)
12. **Heapsort's time and extra space?** $\Theta(n\lg n)$ always, $O(1)$ extra — in place. (Ch. 5)

13. **What does randomizing quicksort's pivot buy?** Expected $\Theta(n\lg n)$ on every input; the $\Theta(n^2)$ worst case remains, but no adversary can force it. (Ch. 5)
14. **The comparison-sort lower bound, and why?** $\Omega(n\lg n)$: a decision tree with $n!$ leaves has height $\geq \lg(n!) = \Theta(n\lg n)$. (Ch. 5)
15. **Why is a BST's inorder walk sorted?** Left subtree $<$ root $<$ right subtree, so left–root–right emits increasing keys; induction. (Ch. 6)
16. **The AVL height bound, and where it comes from?** $h \leq 1.4405\lg(n+2) - 0.3277$, from $N(h) = 1 + N(h-1) + N(h-2)$. (Ch. 6)
17. **How many rotations can one AVL insertion cause? One deletion?** At most two; up to $\Theta(\lg n)$. (Ch. 6)
18. **What makes a problem fit dynamic programming rather than divide and conquer?** Overlapping subproblems plus optimal substructure. (Ch. 7)
19. **Memoization against tabulation, in one line?** Both store each subproblem once; memoization keeps the recursion and fills on demand, tabulation removes it and fills in dependency order. (Ch. 7)
20. **The four steps of designing a dynamic program?** Characterize the optimal structure, write the recurrence, compute bottom-up, reconstruct from the recorded choices. (Ch. 7)
21. **What makes an unfinished proof still useful?** Its structure: the claim, the technique, the base case, a precise hypothesis, the steps that are justified, and an honest label on the gap. (Ch. 8, Section 16.2.2)
22. **The two hypotheses a greedy proof needs?** The greedy-choice property and optimal substructure. (Ch. 9)
23. **Why does greedy solve fractional knapsack but not 0/1?** Fractional lets you fill the capacity exactly with the best ratio; 0/1 cannot split an item, so a locally best choice can waste capacity. (Ch. 9)
24. **In one line, why is Huffman optimal?** The two lowest-frequency symbols can be made deepest siblings without loss (greedy choice), and merging them leaves a smaller instance of the same problem (optimal substructure). (Ch. 9, Problem 16.16)
25. **BFS and DFS differ in what, exactly?** The container: a FIFO queue gives fewest-edge order; a stack, explicit or recursive, gives depth-first order. (Ch. 10)
26. **What does DFS give you that BFS does not?** Discovery and finish times, and hence back edges, cycle detection, topological sort and strongly connected components. (Ch. 10, Problem 16.8)
27. **Adjacency list or adjacency matrix?** List for sparse graphs ($\Theta(V+E)$ space, $\Theta(\deg u)$ to scan u); matrix for dense graphs or when "is (u, v) an edge?" must be $\Theta(1)$. (Ch. 10)
28. **State the cut property.** For any cut respected by the current forest, a lightest edge crossing it is safe to add. (Ch. 11)
29. **Prim or Kruskal?** Prim on dense graphs ($\Theta(V^2)$ with an array); Kruskal on sparse ones, or when the edges are already sorted. (Ch. 11)
30. **Does a topological sort exist for every directed graph?** No — exactly for the acyclic ones; a cycle of any length forbids it. (Ch. 10, 11)
31. **What does an engineering evaluation of an implemented algorithm need?** A stated input distribution, measured times, a comparison against a baseline, and a conclusion that names the trade-off. (Ch. 12)
32. **What is relaxation?** `if d[v] > d[u] + w(u,v) then d[v] = d[u] + w(u,v)` — it can only lower an estimate. (Ch. 13, Equation (16.1))

33. **Why does Dijkstra fail on negative edges?** It finalizes a vertex on extraction, and a later negative edge could have improved it. (Ch. 13, Section 16.7.7)
34. **Why $|V| - 1$ passes in Bellman–Ford, and what does one more pass do?** A shortest path with no negative cycle has at most $|V| - 1$ edges, and pass i makes every i-edge path correct; a further pass that still relaxes an edge proves a negative cycle. (Ch. 13, Problem 16.7)
35. **The three amortized techniques?** Aggregate (total over n operations, divided by n), accounting (assign fees and keep a non-negative credit), potential (define Φ and use Equation (16.6)). (Ch. 14)
36. **Why is a doubling dynamic table $O(1)$ amortized per insertion?** n insertions cost $n + \sum_j 2^j < 3n$. (Ch. 14, Problem 16.10)
37. **What condition must a potential function satisfy?** $\Phi(D_i) \geq \Phi(D_0)$ for all i; usually one takes $\Phi(D_0) = 0$ and $\Phi \geq 0$. (Ch. 14, Problem 16.10(f))
38. **Floyd–Warshall's recurrence, in words?** The shortest $i \to j$ path using only $\{1..k\}$ as interior vertices either avoids k or goes $i \rightsquigarrow k \rightsquigarrow j$. (Ch. 15, Equation (16.10))
39. **Why does Johnson's reweighting preserve shortest paths?** The potentials telescope: every $u \rightsquigarrow v$ path shifts by the same $h(u) - h(v)$. (Ch. 15, Equation (16.12))
40. **The one question to ask before writing any algorithm?** What is the shape of the input, and what is the shape of the answer — because those two together name the algorithm. (Ch. 16, Section 16.14)

16.14 The Algorithm Selection Table

This is the page to keep. Every row is a *shape* — a way the problem looks when you first read it — and the algorithm that shape calls for, with the reason and the cost. It is deliberately arranged by shape rather than by algorithm, because at the moment you need it you know the problem and not yet the answer.

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	$\Theta(n\lg n)$
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	$\Theta(V+E)$

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
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	$O(E\lg V)$
Shortest path, one source, some weights negative	Bellman–Ford	Relaxes everything $\lvert V\rvert - 1$ times; detects negative cycles	$\Theta(VE)$
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	$O(V^2\lg V + VE)$
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	$O(\alpha(n))$ / $O(\lg n)$

Two rules for using **Table 16.19** honestly.

First, the shape comes from the input and the required output, not from the words of the problem. "Find the cheapest way to connect all the villages" and "wire a chip with the least copper" are the same row. "Find the shortest route visiting every village once" looks similar and is a different row entirely — it is the second-to-last row, the NP-hard one.

Second, a row is a starting point, not a verdict. The cost column is asymptotic, and statement (12) of Problem 16.1 is a permanent standing objection to reading it as gospel. Measure before you commit.

16.15 Where to Go Next

This book covers one semester's worth of the subject, and the subject is much larger. Six large territories were left out, and the point of this section is that you are now equipped to enter every one of them. Each paragraph says what the territory is, why it was out of scope, what you already own that will carry you into it, and one book to start with.

Network flow. A directed graph whose edges have *capacities*, a source and a sink; the question is how much can be pushed from one to the other. The Ford–Fulkerson method finds a maximum flow by repeatedly pushing along an augmenting path in the residual graph; the max-flow min-cut theorem says the answer equals the capacity of the lightest cut separating source from sink, which is one of the most beautiful dualities in the subject. It was left out because it needs a full chapter on residual graphs before anything can be proved. You already own everything else: BFS finds the augmenting path (that is Edmonds–Karp, and the BFS is why it terminates in $O(VE^2)$), and the cut vocabulary is Chapter 11's. Bipartite matching, project selection, image segmentation and the vertex-cover bound of Problem 16.20 are all flow problems in disguise. Start with [5,

ch. 7], which develops flow from scratch and then spends a hundred pages on the problems that reduce to it; then [1, ch. 24].

NP-completeness and approximation. Problem 16.4 needed a 2-approximation because the exact problem is intractable, but it never said what *intractable* means. The theory: a problem is in NP if a proposed solution can be checked in polynomial time; it is NP-complete if every problem in NP reduces to it in polynomial time; and nobody knows whether $\mathrm{P} = \mathrm{NP}$, though almost everyone believes not. Traveling salesman, 0/1 knapsack, vertex cover, Hamiltonian cycle, graph coloring and boolean satisfiability are all NP-complete, and you have met four of the six in this book. The skill to acquire is **reduction**: showing that a new problem is at least as hard as a known one — Section 16.6.6 did one, from Hamiltonian cycle to general TSP. Then comes the constructive half — approximation algorithms, with proved ratio bounds of the kind you wrote in Problems 16.4 and 16.20. The classical catalog is [6]; the modern treatment of the approximation side is [7]; [1, ch. 34–35] is the compact version.

String matching. Given a text of length n and a pattern of length m, find every occurrence. The naive method is $\Theta(nm)$; Knuth–Morris–Pratt [8] achieves $\Theta(n + m)$ by precomputing, for every prefix of the pattern, the length of its longest proper prefix that is also a suffix — so that a mismatch never sends the text pointer backward. Rabin–Karp does it with rolling hashes; Boyer–Moore is usually fastest in practice by scanning the pattern backward. Beyond matching lie suffix trees and suffix arrays, which answer "where does this substring occur?" for *any* query substring after linear preprocessing. You already own the two prerequisites: the automaton idea is a state machine over the pattern, and the analysis is amortized — the KMP proof is an aggregate argument of exactly the shape of the binary counter of Problem 16.9. The standard reference, still unmatched for depth, is [9]; [1, ch. 32] is the introduction.

Computational geometry. Points, segments, polygons: does this segment cross that one, what is the convex hull of these points, which pair is closest, which point is nearest a query. The techniques are the ones you have — divide and conquer gives closest-pair in $\Theta(n\lg n)$ and is a textbook Master-Theorem application; Graham's scan sorts by angle and then makes one pass with a stack, which is a greedy algorithm with a stack-based amortized analysis; sweep-line algorithms are a priority queue plus a balanced search tree, which is Chapters 5 and 6 combined. The new content is the *predicates*: orientation tests, and the floating-point care they demand. The standard text is [10].

Linear programming. Maximize a linear function of real variables subject to linear inequalities. It subsumes an astonishing amount: shortest paths, maximum flow, and assignment problems are all linear programs, and their duals are the min-cut and potential arguments you have already used — Johnson's potentials h in Problem 16.12 are a feasible solution to the dual of a shortest-path LP, which is why the insight box of Section 16.12 found the same telescoping trick in two chapters. The simplex method is exponential in the worst case and excellent in practice (statement (12) of Problem 16.1, in its purest form); interior-point methods are polynomial. When the variables are required to be integers the problem becomes NP-hard, and *LP relaxation followed by rounding* is one of the main sources of approximation algorithms with proved ratios. A short, readable entry point is [11]; [1, ch. 29] gives the algorithmic core.

Randomized and online algorithms. Two separate ideas that belong together because both replace "worst case over inputs" with something more useful. A **randomized** algorithm makes coin flips: randomized quicksort in Chapter 5 was your first, and the lesson generalizes — randomization converts "bad input" into "bad luck", which an adversary cannot arrange. Beyond it lie randomized selection, treaps and skip lists, hashing analysis, Karger's min-cut, and Monte

Carlo methods with one-sided error. An **online** algorithm must answer before the whole input arrives — a cache deciding what to evict, a server deciding which lift to send — and is judged by its *competitive ratio* against an offline optimum that knew the future. That ratio is the same kind of object as the $\rho(n) = 2$ of Problem 16.4, which is why the two subjects sit together. Start with [12] for randomization and [13] for the online theory.

A seventh remark, not a territory: **the habit of measuring.** Every asymptotic statement in this book is a claim about large n, and every one of them can be checked on your own machine in ten minutes. [14] is the best single book on the practice of choosing and tuning algorithms, and its catalog of problems — "I have this, I want that, here is what is known" — is the professional version of **Table 16.19**.

16.16 Profitina in Practice

This chapter's subject is *choosing*, and choosing is what an engineering team spends its algorithm time on. Profitina — the AI-visibility (GEO, *Generative Engine Optimization*) platform live at `app.profitina.com` — is a useful place to watch it happen. It scans a brand's website and scores how ready the site is to be *cited by an AI answer engine*, from real signals such as content extractability, structured data and technical credibility, and then helps fix what it finds. Almost every row of **Table 16.19** appears somewhere in the product, and in each case the reason for the choice is the reason this book gave.

The crawl queue is a priority queue. Scanning a site means fetching pages, and pages are not equal: the home page and the most linked-to pages are worth fetching first, because a useful score can be reported before the crawl finishes. That is `Extract-Min` on a heap keyed by an estimated priority — Chapter 5's row of the table, chosen because the operation done most often is "give me the most valuable unfetched page".

The site is a graph, and depth is a shortest-path distance. A crawl starts at one URL and follows links. How many clicks a page lies from the home page is its shortest-path distance in an unweighted graph, which is exactly what breadth-first search computes — the degenerate case of Dijkstra in which every edge weighs 1 and the priority queue becomes a plain queue. BFS order is chosen deliberately over depth-first: a crawl that is cut short by a budget has then covered the *shallow* pages, which are the ones an answer engine is most likely to reach. That is drill 25 of Section 16.13, used as a product decision.

The re-scan cadence is a scheduling problem. Each brand has a subscription tier that sets how often it is re-scanned automatically, and the scheduler must fit the day's scans into the available capacity, with alerts when a score moves. Keeping the next-due scan at the front is again a priority queue, keyed by time; choosing which scans fit into a busy hour is the activity-selection row of the table.

Deduplicating near-identical pages is hashing plus string matching. Many sites serve the same content at several URLs. Canonicalizing the URL is a hash-table problem; detecting that two *different* URLs return substantially the same text is a string problem — one of the places where the string-matching territory of Section 16.15 earns its keep.

The weekly score trend is an amortized structure. The historical-analytics dashboard keeps a weekly score trend per brand, exportable as CSV, and that history grows by one row per brand per scan, forever. A table that holds it and doubles when it fills pays exactly the price computed in Problem 16.10: $O(1)$ amortized per insertion, with an occasional expensive copy that must not

land inside a user's request. Problem 16.10(f) is the warning that goes with it — an analysis that "proves" a better bound by letting its potential go negative is borrowing against the future, and the future always sends the bill.

Checking twice is a release gate. Every answer in this chapter was confirmed a second way: Floyd–Warshall against Johnson, the approximation against brute force, three amortized ledgers against one another. Profitina applies the same discipline at scale: the full automated test suite, well over a thousand tests, must pass before any release, and a release reports success only after the deployed version answers its health check.

One rule governs how all of this is used: the platform *advises*; the customer *decides* — a score is an estimate from measured signals, never a guarantee of being cited.

16.16.1 The worked case study: Profitina Laundry

Profitina Laundry is the teaching business of this book — a neighborhood laundry with one press machine, one van and six pickup points, not the real product. On the last night of the book it has three questions, and the whole point is that they are three *different* questions and therefore call for three different algorithms.

- *Which press jobs can one machine finish today?* — a set of intervals, choose the most that do not overlap: **greedy activity selection** (Chapter 9).
- *In what order should the van collect?* — a metric on six points, find a short closed tour: **minimum spanning tree plus a preorder walk**, with the guarantee of Problem 16.4 (Chapter 11).
- *How far is each neighborhood by road?* — a weighted graph with non-negative weights and one source: **Dijkstra** (Chapter 13).

Listing 16.19 answers all three from one data set.

Listing 16.19 The Profitina Laundry nightly planner: three questions from one data set, answered by a greedy algorithm, a minimum spanning tree with a preorder walk, and Dijkstra — one row of **Table 16.19** each.

```
// Profitina Laundry: the nightly planner. One data set, three questions, three
// different algorithms -- chosen by the shape of each question, not by habit.
//   Q1  which press jobs fit on one machine?      greedy activity selection (Ch. 9)
//   Q2  in what order should the van collect?     MST + preorder walk (Ch. 11, 16.6)
//   Q3  how far is each drop point by road?       Dijkstra (Ch. 13)
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
using namespace std;

struct Job { string id; int s, f; };

const int N = 6;
const string pt[N] = {"Depot","Keputih","Gebang","Semolowaru","Manyar","Klampis"};
const int  PX[N] = {0, 4, 8, 6, 2, 0};
const int  PY[N] = {0, 0, 1, 5, 6, 3};

int dist2[N][N];                                  // straight-line (Manhattan) metric
vector<int> kid[N];
vector<int> tourOrder;

void preorder(int u) { tourOrder.push_back(u); for (int v : kid[u]) preorder(v); }

string hhmm(int m) {                              // minutes after 08:00
    char b[16]; snprintf(b, sizeof b, "%02d:%02d", 8 + m / 60, m % 60); return b;
}

int main() {
    // ---------- Q1: greedy activity selection ----------
    vector<Job> job = {{"J1",0,60},{"J2",30,90},{"J3",60,110},{"J4",45,150},
                       {"J5",115,165},{"J6",150,210},{"J7",100,140},{"J8",170,240}};
    sort(job.begin(), job.end(), [](const Job& a, const Job& b) { return a.f < b.f; });
```

Listing 16.19 (Continued) *The Profitina Laundry nightly planner: three questions from one data set, answered by a greedy algorithm, a minimum spanning tree with a preorder walk, and Dijkstra — one row of Table 16.19 each.*

```
    cout << "Q1  one press machine, jobs sorted by finishing time\n";
    int last = -1, chosen = 0;
    for (auto& j : job) {
        bool take = j.s >= last;
        cout << "    " << j.id << "  " << hhmm(j.s) << "-" << hhmm(j.f)
             << (take ? "   TAKE" : "   skip (overlaps)") << "\n";
        if (take) { last = j.f; ++chosen; }
    }
    cout << "    jobs scheduled = " << chosen << "\n\n";

    // ---------- Q2: minimum spanning tree, then a 2-approximate tour ----------
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            dist2[i][j] = abs(PX[i] - PX[j]) + abs(PY[i] - PY[j]);

    vector<int> key(N, INT_MAX), par(N, -1);
    vector<bool> in(N, false);
    key[0] = 0;
    int mst = 0;
    for (int it = 0; it < N; ++it) {
        int u = -1;
        for (int v = 0; v < N; ++v) if (!in[v] && (u == -1 || key[v] < key[u])) u = v;
        in[u] = true; mst += key[u];
        for (int v = 0; v < N; ++v)
            if (!in[v] && dist2[u][v] < key[v]) { key[v] = dist2[u][v]; par[v] = u; }
    }
    for (int v = 0; v < N; ++v) if (par[v] != -1) kid[par[v]].push_back(v);
    for (int v = 0; v < N; ++v) sort(kid[v].begin(), kid[v].end());
    preorder(0);

    int tour = 0;
    for (int i = 0; i < N; ++i) tour += dist2[tourOrder[i]][tourOrder[(i + 1) % N]];
    cout << "Q2  collection route (MST weight " << mst
         << ", so no tour is shorter than " << mst << ")\n    ";
    for (int i = 0; i < N; ++i) cout << pt[tourOrder[i]] << (i + 1 < N ? " -> " : " -> ");
    cout << pt[0] << "\n    full walk costs 2 x MST = " << 2 * mst
         << "; dropping repeats gives " << tour
         << " blocks, and the theorem of 16.6 guarantees\n"
         << "    this is at most twice the unknown optimum\n\n";

    // ---------- Q3: shortest road distances from the depot ----------
    vector<vector<pair<int,int>>> road(N);
    auto link = [&](int a, int b, int w) { road[a].push_back({b, w}); road[b].push_back({a,
w}); };
    link(0,1,4); link(0,5,3); link(1,2,5); link(2,3,6); link(3,4,5);
    link(4,5,5); link(1,3,7); link(1,5,7); link(0,4,8);
```

Listing 16.19 (Continued) *The Profitina Laundry nightly planner: three questions from one data set, answered by a greedy algorithm, a minimum spanning tree with a preorder walk, and Dijkstra — one row of Table 16.19 each.*

```
    vector<int> d(N, INT_MAX);
    d[0] = 0;
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    Q.push({0, 0});
    while (!Q.empty()) {
        auto [du, u] = Q.top(); Q.pop();
        if (du > d[u]) continue;                    // stale queue entry
        for (auto [v, w] : road[u])
            if (d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
    }
    cout << "Q3  shortest road distance from the depot\n";
    vector<int> ord(N); for (int i = 0; i < N; ++i) ord[i] = i;
    sort(ord.begin(), ord.end(), [&](int a, int b) { return d[a] < d[b]; });
    for (int v : ord)
        cout << "    " << setw(12) << left << pt[v] << right << setw(4) << d[v] << "
blocks\n";
    return 0;
}
```

Compiled with `g++ -std=c++17 -O2 -Wall` and run:

```
Q1  one press machine, jobs sorted by finishing time
    J1  08:00-09:00   TAKE
    J2  08:30-09:30   skip (overlaps)
    J3  09:00-09:50   TAKE
    J7  09:40-10:20   skip (overlaps)
    J4  08:45-10:30   skip (overlaps)
    J5  09:55-10:45   TAKE
    J6  10:30-11:30   skip (overlaps)
    J8  10:50-12:00   TAKE
    jobs scheduled = 4

Q2  collection route (MST weight 22, so no tour is shorter than 22)
    Depot -> Keputih -> Gebang -> Klampis -> Manyar -> Semolowaru -> Depot
    full walk costs 2 x MST = 44; dropping repeats gives 40 blocks, and the theorem of 16.6
guarantees
    this is at most twice the unknown optimum

Q3  shortest road distance from the depot
    Depot          0 blocks
    Klampis        3 blocks
    Keputih        4 blocks
    Manyar         8 blocks
    Gebang         9 blocks
    Semolowaru    11 blocks
```

Notice what the greedy schedule does with J4: it runs from 08:45 to 10:30, which is long, and taking it would block both J3 and J5. Sorting by *finishing* time rather than by start time or by duration is what makes the algorithm skip it, and the greedy-choice proof of Chapter 9 is what makes that skip provably right rather than merely plausible. Four jobs is optimal, not just good. And look at the distances from the depot in Q3: 0, 3, 4, 8, 9, 11 blocks — exactly the Manhattan distances of Q2's grid for the same six points. That is a coincidence of this road network, not a law: Q2 was computed from coordinates and Q3 from nine road segments, which are two different graphs. Close the road from the depot to Klampis and Q3 changes while Q2 does not. Two questions about "distance" were questions about two different graphs, and each got its own algorithm.

16.17 Common Mistakes, and How to Avoid Them

1. **Answering the statement you expected instead of the one that is written.** Underline the verb and the qualifier — *linear search*, *at least*, *always*, *not possible* — before deciding (Section 16.3.12).
2. **Giving an O bound without naming the case.** "Quicksort is $O(n^2)$" is true of the worst case and misleading everywhere else. Say *worst*, *average* or *expected*, every time.
3. **Confusing a subsequence with a substring.** A subsequence keeps order but not adjacency. State this in your assumptions before designing anything (Section 16.4.10).
4. **Computing the value of a dynamic program and not the solution.** Keep the predecessor array and walk it back. "Find the longest increasing subsequence" means produce it.
5. **Using weak induction where strong induction is needed.** If the inductive step splits an object of size n into pieces of sizes k and $n-1-k$, the hypothesis must cover *every* smaller size (Section 16.5.8).
6. **Asserting a step that needs a hypothesis.** "Shortcutting makes the walk shorter" is only true under the triangle inequality. Name the hypothesis where you use it (Section 16.6.10).
7. **Proving a ratio bound without a lower bound on the optimum.** Every such proof has the same skeleton: find something that is at most the optimum ($d(T)$, $|A|$), and show the answer is at most twice it.
8. **Treating Dijkstra as universal.** It is greedy, and greedy is safe here only because extending a path cannot shorten it. One negative edge and the finalization is wrong (Section 16.7.7). And a correct answer on one graph with negative edges, as in Problem 16.8, proves nothing.
9. **Extracting the wrong vertex in a Dijkstra trace.** The next vertex is the unfinished one with the smallest *current* estimate, including estimates lowered in the previous iteration — not the one that was smallest when it was first reached. Write the whole row of **Table 16.8** before choosing.
10. **Forgetting Bellman–Ford's closing check, or stopping it too soon.** Without the extra sweep there is no negative-cycle detection; and an early exit is legal only after a pass with *no* change, not after a pass with few changes (Problem 16.7).
11. **Taking the topological order from increasing finish times.** The order is *decreasing* finish time; increasing finish time lists every edge backward (Problem 16.8).
12. **Letting $\infty + w$ overflow.** In code, ∞ is a large number, and adding a negative weight to it makes a large number that is no longer "infinite". Guard every relaxation with `d[u] < INF`, as every listing in this chapter does.
13. **Choosing a potential function without checking $\Phi(D_i) \geq \Phi(D_0)$.** A potential that goes negative "proves" bounds that are false — Problem 16.10(f) "proved" $2n$ for a table whose true cost at $n = 17$ is 48, not 34.
14. **Confusing amortized with average-case cost.** An amortized bound holds for *every* sequence, in the worst case, with no probability anywhere. An average-case bound depends on an input distribution.
15. **Putting k in the wrong loop of Floyd–Warshall.** The intermediate vertex k must be the *outermost* loop; with k innermost, d_{ij} is finalized before the routes it depends on are known (Section 15.6.6).
16. **Forgetting to undo Johnson's reweighting.** Dijkstra on w' returns $\hat{d}(u,v)$, not $\delta(u,v)$; the answer is $\hat{d}(u,v) - h(u) + h(v)$ (Problem 16.12).

17. **Adding a constant to every edge to remove negative weights.** A path with more edges gains more, so the shortest path can change (Exercise 1(d)). Johnson's potential works precisely because its correction telescopes to a quantity that depends only on the endpoints.
18. **Calling a pseudo-polynomial algorithm polynomial.** $\Theta(nW)$ knapsack is exponential in the *size* of the input (Problem 16.19).
19. **Illustrating a graph instead of an algorithm.** An illustration must show the estimates changing. Two intermediate states minimum.
20. **Reading Table 16.19 as a verdict rather than a starting point.** The cost column is asymptotic; constants, memory and input size still decide (statement (12) of Problem 16.1).

16.18 Chapter Summary, and a Closing Word

This chapter consolidated Chapters 13 to 15 and integrated the whole book by working twenty-one problems, each derived from first principles, each checked by a program that was compiled and run.

The substantive results, gathered:

- **Problem 16.1.** Nine true, six false. The false ones are (3) — linear search is $\Theta(n)$ whether or not the array is sorted; (8) — insertion into an ordered list is $\Theta(n)$ because the cheap half does not rescue the expensive half; (11) — O is an upper bound, not a lower one; (12) — constants, memory, input size and stability all outrank asymptotics sometimes; (13) — depth-first and breadth-first order agree on 26 of the 38 four-vertex graphs, and on a three-vertex path.
- **Problem 16.2.** $L[i]$, the length of the longest increasing subsequence *ending at* $S[i]$, satisfies Equation (16.2). Two nested loops give $\Theta(n^2)$; a predecessor array gives the subsequence itself; a sorted `tail` array with binary search gives $O(n\lg n)$.
- **Problem 16.3.** $X(n) = n + 1$ by strong induction on the number of internal nodes, splitting at the root, Equation (16.3); k cancels, which is why the result is shape-independent.
- **Problem 16.4.** $d(T) \leq d(L^*)$ because deleting an edge from the optimal tour leaves a spanning tree; $d(W) = 2d(T)$ because a full walk uses every tree edge twice; $d(L) \leq d(W)$ by the triangle inequality; together, Equation (16.5). On six real points the observed ratio was $40/28 = 1.4286$.
- **Problem 16.5.** The two algorithms differ only in the order and multiplicity of relaxations; that difference is why routers use one and maps use the other.
- **Problems 16.6 to 16.8.** Dijkstra finalizes vertices in increasing order of distance and relaxes each edge once; Bellman–Ford needs up to $|V| - 1$ passes, fewer with a lucky edge order, and one more pass that still relaxes something proves a negative cycle, which the predecessor array then exhibits; in a DAG, one pass in topological order suffices for any weights.
- **Problems 16.9 and 16.10.** The binary counter costs less than $2n$ and the doubling table less than $3n$ for n operations, by the aggregate, accounting and potential methods alike; the accounting credit and the potential are the same number, and a potential that may go negative proves false bounds.
- **Problems 16.11 and 16.12.** Floyd–Warshall computes $D^{(0)}, \ldots, D^{(n)}$ by Equation (16.10) in $\Theta(V^3)$; Johnson reweights by Equation (16.11), whose telescoping preserves every shortest path, and reaches the same matrix with V runs of Dijkstra.
- **Problems 16.13 to 16.21** revisited the rest of the book: longest paths in a DAG, the edge count of a tree, Huffman's optimality, the two all-pairs algorithms compared, pseudo-polynomial knapsack, the vertex-cover 2-approximation, and Prim against Kruskal.

Around the problems, the chapter left you a method for multi-part problems and stalled proofs, forty self-check drills across all sixteen chapters, an algorithm selection table organized by the shape of the problem rather than by the name of the algorithm, and a map of six territories — network flow, NP-completeness and approximation, string matching, computational geometry, linear programming, and randomized and online algorithms — that you are now equipped to enter.

A closing word

Sixteen chapters ago this book began with a becak driver and a siomay seller, and the promise that someone who knew nothing could start at page one and finish able to implement the material. If you have arrived here, that promise has been kept on your side of the bargain, which is the harder side.

It is worth being precise about what you now have, because it is not what most people think a book like this gives.

You do not have a catalog of algorithms. You could look those up. What you have is the ability to look at a problem you have never seen — a list of intervals, a road network with a negative toll, a table that must grow without limit, a proof that will not come out at the first attempt — and *recognize its shape*. Problem 16.4 is the evidence. This book never taught approximation algorithms. You were handed a problem from a subject it did not cover, and everything you needed was already in your hands: a spanning tree, a walk, and one inequality. That is not luck. It is what a real education in algorithms is for.

You also have something rarer, which is the habit of not believing yourself. Every distance, every tree weight, every matrix and every amortized bound in this chapter was computed twice — once by hand and once by a program — and they agreed. When one day they do not agree, you will know that one of the two is wrong and that finding out which is the job. That habit will outlast every asymptotic bound in this book.

And you have the arithmetic of consequence. A minimum spanning tree costs no more than the best tour. A doubling table costs less than three per insertion, and a table that grows by a constant costs a quadratic. One negative edge is enough to make a greedy choice unsafe, and one potential that dips below zero is enough to make a proof false. Algorithms is the discipline in which vague feelings about cost are replaced by numbers you can defend, and that replacement, more than any single technique, is the thing worth carrying out of this book.

Good luck — and stay awesome. The rest of it is now yours to do.

16.19 Exercises

1. Mark each statement True or False and give a one-sentence justification. (a) Insertion into a binary heap of n elements costs $O(\lg n)$. (b) Every directed graph with no cycles has at least one vertex of in-degree zero. (c) If every edge weight of a connected graph is increased by the same constant $c > 0$, the minimum spanning tree is unchanged. (d) If every edge weight is increased by the same constant $c > 0$, every shortest path is unchanged. (e) A greedy algorithm that is optimal for all inputs of size ≤ 10 is optimal for all inputs.
2. Run **Pseudocode 16.1** by hand on $S = \langle 4,12,1,9,5,15,7,11 \rangle$. Give the L and P arrays and the subsequence the backtrack produces.

3. Prove that an extended binary tree with n internal nodes has $2n + 1$ nodes in total, and deduce how many null pointers the *original* binary tree contains.
4. Take the six points $A = (3,0)$, $B = (3,2)$, $C = (0,3)$, $D = (0,6)$, $E = (0,0)$, $F = (3,5)$ with the Manhattan metric. Build the minimum spanning tree rooted at A, give the preorder tour and its length, find the optimal tour, and report the ratio. Compare with the 1.4286 of Section 16.6.7.
5. Trace Dijkstra from A on the digraph with edges $A \to B$ (7), $A \to C$ (9), $A \to F$ (14), $B \to C$ (10), $B \to D$ (15), $C \to D$ (11), $C \to F$ (2), $F \to E$ (9), $D \to E$ (6). Give the extraction order, every estimate change, and the shortest-path tree.
6. Consider the path $s \to a \to b \to c$ with every edge of weight 1. How many passes does Bellman–Ford need if the edges are relaxed in the order $(s, a), (a, b), (b, c)$? In the order $(b, c), (a, b), (s, a)$? What does this say about the $|V| - 1$ bound?
7. Construct a four-vertex instance that does **not** satisfy the triangle inequality and on which the algorithm of Problem 16.4 returns a tour more than twice the optimum. Say which step of the proof fails.
8. Six Profitina Laundry service tags occur with frequencies W: 40, D: 25, P: 15, F: 10, S: 6, I: 4 in a hundred-tag sample. Build the Huffman code, give the codeword lengths, compute the total bits, and compare with a fixed-length code.
9. A stack supports `PUSH`, `POP` and `MULTIPOP(k)`, which pops $\min(k, |S|)$ items. Using the potential function $\Phi(S) = |S|$, show that the amortized cost of each of the three operations is $O(1)$, and hence that any sequence of n operations costs $O(n)$.
10. For each problem shape, name the algorithm from **Table 16.19**, give its cost, and say in one clause why. (a) A courier must visit 12 addresses and return, minimizing distance, and distances are straight-line. (b) A build system must order 400 compilation units with dependencies. (c) A spell-checker must decide whether each of 10^6 words is in a fixed dictionary of 10^5 words. (d) A game must find the cheapest path across a grid where some tiles restore health (negative cost). (e) A network monitor must report the distance between every pair of 200 routers.
11. Compute shortest paths from s in the DAG with edges $s \to a$ (2), $s \to b$ (6), $a \to b$ (3), $a \to c$ (7), $b \to c$ (−4), $b \to d$ (2), $c \to d$ (1), giving the topological order and d/π after each vertex is processed.
12. A binary counter supports `INCREMENT` and `RESET`, where `RESET` sets every bit to 0 and costs the number of bits that were 1. Using a potential function, show that any sequence of n operations starting from zero costs at most $2n$.
13. Run Floyd–Warshall on the three-vertex graph with edges $1 \to 2$ (5), $1 \to 3$ (6), $2 \to 3$ (−1), $3 \to 1$ (2). Give $D^{(0)}$ to $D^{(3)}$.
14. For the graph of Exercise 13, compute Johnson's potentials h and the reweighted edges, and use them to recover $\delta(1,3)$ from a Dijkstra run from vertex 1. Then explain, with a number from this graph, why adding the constant 3 to every edge would not be a safe alternative.

16.20 Solutions to the Exercises

1. (a) **True** — place the new element at the end and sift it up the $\Theta(\lg n)$-deep path. (b) **True** — a DAG has a topological order, and its first vertex has in-degree zero; equivalently, follow incoming edges backward and acyclicity forces the walk to stop. (c) **True** — every spanning tree has exactly $V - 1$ edges, so every spanning tree's weight increases by exactly $(V - 1)c$ and the ordering of spanning trees by weight is unchanged. (d) **False** — different paths have different numbers of edges, so a k-edge path gains kc while a path with fewer edges gains less; adding a constant can

therefore change which path is shortest. This is why Johnson reweights with a telescoping potential and not with a constant (Exercise 14). (e) **False** — optimality on small inputs proves nothing about large ones; the 0/1 knapsack greedy is optimal on many small instances and wrong in general.

2. Working the outer loop: $L[1] = 1, P[1] = 0$; $L[2] = 2, P[2] = 1$ (since $4 < 12$); $L[3] = 1, P[3] = 0$; $L[4] = 2, P[4] = 1$ ($4 < 9$); $L[5] = 2, P[5] = 1$ ($4 < 5$); $L[6] = 3, P[6] = 2$ ($12 < 15$ with $L[2] = 2$); $L[7] = 3, P[7] = 5$ ($5 < 7$ with $L[5] = 2$); $L[8] = 4, P[8] = 7$ ($7 < 11$ with $L[7] = 3$).

Table 16.20 The tables for Exercise 2. The maximum is $L[8] = 4$, so the backtrack starts at $k = 8$.

i	1	2	3	4	5	6	7	8
$S[i]$	4	12	1	9	5	15	7	11
$L[i]$	1	2	1	2	2	3	3	4
$P[i]$	0	1	0	1	1	2	5	7

Table 16.20 collects them. Backtrack from $k = 8$: $11 \to P[8] = 7 \to 7 \to P[7] = 5 \to 5 \to P[5] = 1 \to 4 \to P[1] = 0$. Reversed: $\langle 4,5,7,11 \rangle$, of length 4. (Note that $\langle 4,9,15 \rangle$ has length 3 and $\langle 1,5,7,11 \rangle$ also has length 4 — the answer is not unique.)

3. Every internal node of an extended binary tree has exactly two children and every external node has none, and by Section 16.5.3 there are n internal and $n + 1$ external nodes, so the total is $n + (n + 1) = 2n + 1$ — always odd. The external nodes are exactly the null pointers of the original tree, so the original binary tree with n nodes contains $n + 1$ null pointers. Check on a single node: $n = 1$, two null children, $1 + 1 = 2$. Check on **Figure 16.1**: $n = 4$, five nulls, total $2(4) + 1 = 9$ nodes in the extended tree — four circles and five squares.

4. Manhattan distances: $AB = 2$, $AC = 6$, $AD = 9$, $AE = 3$, $AF = 5$, $BC = 4$, $BD = 7$, $BE = 5$, $BF = 3$, $CD = 3$, $CE = 3$, $CF = 5$, $DE = 6$, $DF = 4$, $EF = 8$. Prim from A takes $A - B$ (2), then $A - E$ (3), then $E - C$ (3), then $C - D$ (3), then $B - F$ (3): $d(T) = 2 + 3 + 3 + 3 + 3 = 14$. Rooted at A the children are $A \to \{B, E\}$, $B \to \{F\}$, $E \to \{C\}$, $C \to \{D\}$, so the preorder list is A, B, F, E, C, D and the tour costs $2 + 3 + 8 + 3 + 3 + 9 = 28$. Exhaustive search over the 60 tours gives the optimum $A \to B \to F \to D \to C \to E \to A$ at $2 + 3 + 4 + 3 + 3 + 3 = 18$. The ratio is $28/18 = 1.5556$ — worse than Section 16.6.7's 1.4286, and still comfortably under the guaranteed 2. The expensive step is $F \to E$ at 8: the preorder walk is forced to jump from the bottom of one branch to the top of another, which is exactly where this algorithm loses.

5. Extraction order A, B, C, F, D, E.

- Extract A ($d = 0$): $d[B] = 7$, $d[C] = 9$, $d[F] = 14$.
- Extract B ($d = 7$): $7 + 10 = 17 \not< 9$, no change to C; $d[D] = 7 + 15 = 22$.
- Extract C ($d = 9$): $d[D] = 9 + 11 = 20 < 22$; $d[F] = 9 + 2 = 11 < 14$.
- Extract F ($d = 11$): $d[E] = 11 + 9 = 20$.
- Extract D ($d = 20$): $20 + 6 = 26 \not< 20$, no change.
- Extract E ($d = 20$): no out-edges.

Resulting distances $A = 0$, $B = 7$, $C = 9$, $F = 11$, $D = 20$, $E = 20$; shortest-path tree $B \leftarrow A$, $C \leftarrow A$, $F \leftarrow C$, $D \leftarrow C$, $E \leftarrow F$. Note that both $d[D]$ and $d[F]$ were improved after their first assignment — an estimate is provisional until its vertex is extracted. (D and E tie at 20; either may be extracted first.)

6. In the order $(s, a), (a, b), (b, c)$, **one** pass suffices: the single pass sets $d[a] = 1$, then $d[b] = 2$, then $d[c] = 3$. In the reverse order $(b, c), (a, b), (s, a)$, each pass advances the frontier by exactly

one edge: pass 1 sets only $d[a] = 1$; pass 2 sets $d[b] = 2$; pass 3 sets $d[c] = 3$. With $|V| = 4$ that is $|V| - 1 = 3$ passes, so the bound is attained. The lesson, which Problem 16.7 also showed: $|V| - 1$ is a worst case over edge orderings, not a count of passes actually needed. A good ordering — topological order, when one exists — reduces it to one pass, which is precisely the DAG algorithm of **Pseudocode 16.2**.

7. Take A, B, C, D with $d(A, B) = d(A, C) = d(A, D) = d(B, C) = d(B, D) = 1$ and $d(C, D) = 100$. The triangle inequality fails at $d(C, D) = 100 > d(C, A) + d(A, D) = 2$. Prim from A takes $A - B$, $A - C$, $A - D$, all of weight 1, so $d(T) = 3$; the preorder list is A, B, C, D and the tour costs $1 + 1 + 100 + 1 = 103$. The optimum is $A \to C \to B \to D \to A = 1 + 1 + 1 + 1 = 4$, so the ratio is $103/4 = 25.75$, far above 2. The step that fails is Step 3 of Section 16.6.5: shortcutting from C to D replaced the two-edge sub-path $C \to A \to D$ of cost 2 by the single edge of cost 100, and only the triangle inequality forbids that.

8. Merging the two smallest repeatedly: $4 + 6 = 10$; $10 + 10 = 20$; $15 + 20 = 35$; $25 + 35 = 60$; $40 + 60 = 100$. Codeword lengths follow the depths: W at depth 1, D at 2, P at 3, F at 4, and S, I at 5. Total bits

$$\begin{aligned} 40(1) + 25(2) + 15(3) + 10(4) + 6(5) + 4(5) \\ = 40 + 50 + 45 + 40 + 30 + 20 = 225. \end{aligned}$$

The sum of the internal-node values is $10 + 20 + 35 + 60 + 100 = 225$ — the same number, which is the standard alternative way to compute a Huffman cost. A fixed-length code for six symbols needs $\lceil \lg 6 \rceil = 3$ bits each, so 300 bits; the saving is $(300 - 225)/300 = 25\%$. **Listing 16.20** confirms it.

Listing 16.20 The Huffman code of Exercise 8, with the merge sequence, the codewords, and the comparison against a three-bit fixed-length code.

```
// Exercise 8: build the Huffman code for the Profitina Laundry service tags and
// compare it with a fixed-length code.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <map>
using namespace std;

struct Node { int f; char c; Node *l, *r; };
struct Cmp { bool operator()(Node* a, Node* b) const { return a->f > b->f; } };

void codes(Node* t, string s, map<char,string>& out) {
    if (!t->l && !t->r) { out[t->c] = s.empty() ? "0" : s; return; }
    codes(t->l, s + "0", out);
    codes(t->r, s + "1", out);
}

int main() {
    vector<pair<char,int>> freq = {{'W',40},{'D',25},{'P',15},{'F',10},{'S',6},{'I',4}};
    priority_queue<Node*, vector<Node*>, Cmp> Q;
    for (auto [c, f] : freq) Q.push(new Node{f, c, nullptr, nullptr});

    int internalSum = 0;
    while (Q.size() > 1) {
        Node* a = Q.top(); Q.pop();
        Node* b = Q.top(); Q.pop();
        Node* z = new Node{a->f + b->f, '*', a, b};
        internalSum += z->f;
        cout << "merge " << setw(3) << a->f << " + " << setw(3) << b->f
             << " = " << setw(3) << z->f << "\n";
        Q.push(z);
    }
    map<char,string> code;
    codes(Q.top(), "", code);
```

Listing 16.20 (Continued) *The Huffman code of Exercise 8, with the merge sequence, the codewords, and the comparison against a three-bit fixed-length code.*

```
    int total = 0, n = 0;
    cout << "\n sym  freq  code      bits\n";
    for (auto [c, f] : freq) {
        total += f * (int)code[c].size();
        n += f;
        cout << setw(4) << c << setw(6) << f << setw(7) << code[c]
             << setw(10) << f * (int)code[c].size() << "\n";
    }
    cout << "\nHuffman total          = " << total << " bits\n"
         << "sum of internal nodes  = " << internalSum << " bits (same number)\n"
         << "fixed-length 3 bits    = " << 3 * n << " bits\n"
         << "saving                 = " << 100.0 * (3 * n - total) / (3 * n) << " %\n";
    return 0;
}
```

```
merge   4 +   6 =  10
merge  10 +  10 =  20
merge  15 +  20 =  35
merge  25 +  35 =  60
merge  40 +  60 = 100

 sym  freq  code      bits
   W    40      0        40
   D    25     10        50
   P    15    110        45
   F    10   1110        40
   S     6  11111        30
   I     4  11110        20

Huffman total          = 225 bits
sum of internal nodes  = 225 bits (same number)
fixed-length 3 bits    = 300 bits
saving                 = 25 %
```

9. Take $\Phi(S) = |S|$, the number of items on the stack. It satisfies $\Phi(S) \geq 0 = \Phi(S_0)$, so it is a legal potential. The amortized cost is given by Equation (16.6).

- `PUSH`: actual cost 1; the stack grows by one, so $\Delta\Phi = +1$; $\hat{c} = 1 + 1 = 2$.
- `POP`: actual cost 1; $\Delta\Phi = -1$; $\hat{c} = 1 - 1 = 0$.
- `MULTIPOP(k)`: let $m = \min(k, |S|)$ be the number actually popped. Actual cost m; $\Delta\Phi = -m$; $\hat{c} = m - m = 0$.

Every amortized cost is at most 2, a constant. Since $\Phi(S_n) \geq \Phi(S_0)$, Equation (16.7) bounds the true total by the amortized total:

$$\sum_{i=1}^{n} c_i \;=\; \sum_{i=1}^{n} \hat{c}_i - \Phi(S_n) + \Phi(S_0) \;\leq\; \sum_{i=1}^{n} \hat{c}_i \;\leq\; 2n.$$

So any sequence of n operations costs $O(n)$, even though a single `MULTIPOP` can cost $\Theta(n)$. The intuition the potential encodes: each `PUSH` pays one unit in advance for the eventual removal of the item it pushed, so popping is free when it happens.

10. (a) Twelve addresses with straight-line distances is a **metric TSP**; exact is feasible at $n = 12$ by dynamic programming over subsets in $\Theta(2^n n^2)$, and the MST-plus-preorder 2-approximation of Problem 16.4 runs in $\Theta(n^2)$ with a proved guarantee — use the approximation if the answer is needed repeatedly. (b) Four hundred units with dependencies is a **topological sort**, $\Theta(V + E)$, because the requirement is exactly "every unit after its prerequisites", which can be met precisely when the dependency graph is acyclic. (c) A million membership queries against a fixed dictionary is a **hash table**, expected $\Theta(1)$ per query, because order is never needed; a sorted array with binary search at $\Theta(\lg n)$ is the alternative if memory is very tight. (d) A grid with some negative-cost tiles rules out Dijkstra, so use **Bellman–Ford**, $\Theta(VE)$, which also reports a negative cycle if the healing tiles form a loop that can be farmed forever. (e) All pairs among 200 routers is dense enough that **Floyd–Warshall** at $\Theta(V^3) = 8 \times 10^6$ basic steps is simplest and fastest to write; Johnson would be preferred only if the graph were much larger and sparse.

11. Every edge goes forward in s, a, b, c, d, so that is a topological order. Processing: s — $d[a] = 2$ $(\pi = s)$, $d[b] = 6$ $(\pi = s)$. a — $a \to b$: $2 + 3 = 5 < 6$, so $d[b] = 5$, $\pi[b] = a$; $a \to c$: $d[c] = 9$, $\pi[c] = a$. b — $b \to c$: $5 + (-4) = 1 < 9$, so $d[c] = 1, \pi[c] = b$; $b \to d$: $d[d] = 7, \pi[d] = b$. c — $c \to d$: $1 + 1 = 2 < 7$, so $d[d] = 2, \pi[d] = c$. d — no out-edges. Result $\delta = (s{:}\,0,\ a{:}\,2,\ b{:}\,5,\ c{:}\,1,\ d{:}\,2)$, and the path to d is $s \to a \to b \to c \to d$ with weight $2 + 3 - 4 + 1 = 2$.

12. Let Φ be the number of 1 bits, so $\Phi(D_0) = 0$ and $\Phi \geq 0$. `INCREMENT` is analyzed exactly as in Problem 16.9(c): $\hat{c} \leq 2$. `RESET` on a counter with b one-bits costs b and drops the potential from b to 0, so $\hat{c} = b + (0 - b) = 0$. Every amortized cost is at most 2, so by Equation (16.7) any n operations cost at most $2n$. In accounting terms: the credit left on each 1 bit when it was set pays for clearing it, whether the clearing is done by a later `INCREMENT` or by `RESET`. (For the cost of `RESET` to be the number of 1 bits rather than k, the counter must know where its highest 1 bit is — a single extra index, maintained in $O(1)$ per operation.)

13. $D^{(0)} = \begin{pmatrix} 0 & 5 & 6 \\ \infty & 0 & -1 \\ 2 & \infty & 0 \end{pmatrix}$. Round 1: only $d_{31} = 2$ is finite in column 1, so $d_{32} = \min(\infty, 2 + 5) = 7$ and $d_{33} = \min(0,2 + 6) = 0$: $D^{(1)} = \begin{pmatrix} 0 & 5 & 6 \\ \infty & 0 & -1 \\ 2 & 7 & 0 \end{pmatrix}$. Round 2: only $d_{23} = -1$ is finite in row 2, so $d_{13} = \min(6,5 - 1) = 4$ and $d_{33} = \min(0,7 - 1) = 0$: $D^{(2)} = \begin{pmatrix} 0 & 5 & 4 \\ \infty & 0 & -1 \\ 2 & 7 & 0 \end{pmatrix}$. Round 3: row 3 is $(2,7,\cdot)$, so $d_{21} = \min(\infty, -1 + 2) = 1$, $d_{12} = \min(5,4 + 7) = 5$, $d_{11} = \min(0,4 + 2) = 0$, $d_{22} = \min(0, -1 + 7) = 0$: $D^{(3)} = \begin{pmatrix} 0 & 5 & 4 \\ 1 & 0 & -1 \\ 2 & 7 & 0 \end{pmatrix}$. The diagonal is zero, so there is no negative cycle; the only cycles, $1 \to 2 \to 3 \to 1$ and $1 \to 3 \to 1$, weigh 6 and 8.

14. From the auxiliary vertex q, every h starts at 0; the edge $2 \to 3$ lowers $h(3)$ to -1, and nothing else improves, so $h = (0,0,-1)$. By Equation (16.11): $w'(1,2) = 5 + 0 - 0 = 5$, $w'(1,3) = 6 + 0 + 1 = 7$, $w'(2,3) = -1 + 0 + 1 = 0$, $w'(3,1) = 2 - 1 - 0 = 1$ — all non-negative. Dijkstra from 1 on w' reaches 2 at 5 and then 3 at $\min(7,5 + 0) = 5$, so $\hat{d}(1,3) = 5$ and $\delta(1,3) = 5 - h(1) + h(3) = 5 - 0 - 1 = 4$, agreeing with $D^{(3)}$ of Exercise 13. Now add 3 to every edge instead: the direct edge $1 \to 3$ becomes $6 + 3 = 9$ while the two-edge path $1 \to 2 \to 3$ becomes $(5 + 3) + (-1 + 3) = 10$, so the "shortest" path would switch to the direct edge, whose true weight 6 is worse than the true optimum 4. The constant penalizes paths by their number of edges; the potential penalizes them only by their endpoints.

References

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

[2] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," *Operations Research Forum*, vol. 3, no. 20, 2022. (Originally Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.)

[3] A. R. Karlin, N. Klein, and S. Oveis Gharan, "A (slightly) improved approximation algorithm for metric TSP," in *Proc. 53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, 2021, pp. 32–45. doi: 10.1145/3406325.3451009. Journal version: *Operations Research*, 2023, doi: 10.1287/opre.2022.2338.

[4] L. Gurvits, N. Klein, and J. Leake, "From trees to polynomials and back again: New capacity bounds with applications to TSP," in *Proc. 51st International Colloquium on Automata, Languages, and Programming (ICALP)*, LIPIcs vol. 297, 2024, paper 79. doi: 10.4230/LIPIcs.ICALP.2024.79.

[5] J. Kleinberg and É. Tardos, *Algorithm Design*. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman, 1979.

[7] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2011.

[8] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[9] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge, U.K.: Cambridge Univ. Press, 1997.

[10] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Berlin, Germany: Springer, 2008.

[11] J. Matoušek and B. Gärtner, *Understanding and Using Linear Programming*. Berlin, Germany: Springer, 2007.

[12] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 1995.

[13] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

[14] S. S. Skiena, *The Algorithm Design Manual*, 3rd ed. Cham, Switzerland: Springer, 2020.

[15] *Programming Languages — C++*, ISO/IEC Standard 14882:2017, International Organization for Standardization, Geneva, Switzerland, 2017.

The shortest-path material of Problems 16.5 to 16.8 follows Cormen et al. [1, ch. 22], the amortized analyses of Problems 16.9 and 16.10 follow [1, ch. 16], and the all-pairs algorithms of Problems 16.11, 16.12 and 16.17 follow [1, ch. 23]; the figures **Figure 16.2**, **Figure 16.3**, **Figure 16.4**, **Figure 16.5**, **Figure 16.6**, **Figure 16.7** and **Figure 16.8** are those of Chapter 13, **Figure 16.9** is that of Chapter 14, and **Figure 16.1** illustrates Problem 16.3. The dynamic programs of Problems 16.2 and 16.19, the greedy proof of Problem 16.16 and the graph arguments of Problems 16.3, 16.15 and 16.21 follow [1, ch. 14, 15, 20, 21]; the metric-TSP 2-approximation of Problem 16.4 and the vertex-cover bound of Problem 16.20 are [1, §35.1–35.2], and the 3/2 improvement mentioned in Section 16.6.9 is [2]. The territories of Section 16.15 are introduced in [5] for network flow, [6] and [7] for NP-completeness and approximation, [9] and [8] for string matching, [10] for computational geometry, [11] for linear programming, and [12] and [13] for randomized and online algorithms; [14] is the practitioner's catalog that **Table 16.19** is a one-page version of. Every listing in this chapter conforms to [15] and was compiled with `g++ -std=c++17 -O2 -Wall` and run; the outputs quoted are the outputs produced.