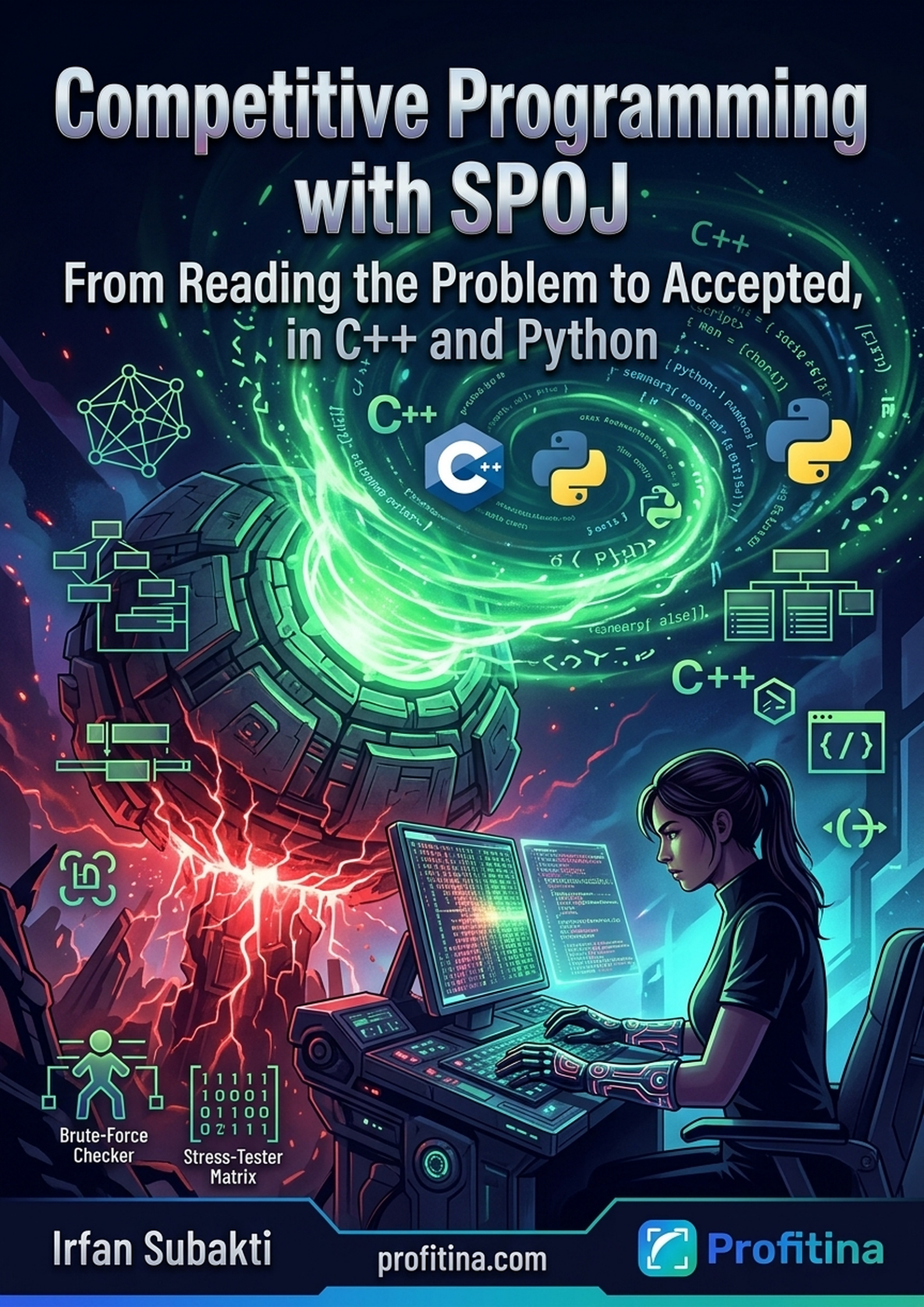
Competitive Programming with SPOJ

From Reading the Problem to Accepted, in C++ and Python



Irfan Subakti

profitina.com

Profitina

Competitive Programming with SPOJ

From Reading the Problem to Accepted, in C++ and Python

Irfan Subakti



Copyright © 2026 Irfan Subakti. 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 and the publisher accept no liability for their use elsewhere.

Contents

Preface ..... **xx**
Who this book is for xx
One idea, two languages xx
What is in it xxi
How to read it xxii
About the author xxii
Acknowledgments xxii

CHAPTER 1 Welcome to Competitive Programming and SPOJ **1**
1.1 A Red Verdict at Midnight 1
1.2 What Competitive Programming Is, and What SPOJ Offers 2
1.3 What the Judge Does With Your Program 3
1.4 How This Book Works 5
1.5 Your Workbench: What We Install and Why 5
1.6 Installing on Windows 10 and 11 6
1.6.1 The C++ compiler, from MSYS2 6
1.6.2 Python 7
1.6.3 VS Code 7
1.7 Installing on macOS 7
1.7.1 The C++ compiler 7
1.7.2 Python and VS Code 8
1.8 Installing on Linux 8
1.9 VS Code: Two Extensions and a Build Task 8
1.10 Your First Programs: Checking the Workbench 9
1.11 Your First SPOJ Problem: TEST 11
1.12 Building a Judge of Your Own 12
1.12.1 Tests as pairs of files 13
1.12.2 The judge 13
1.12.3 Judging correct and wrong programs 16
1.13 Using It for Real 18
1.14 Common Mistakes, and How to Avoid Them 19
1.15 Chapter Summary 19
1.16 Exercises 20
1.17 Solutions to the Exercises 20
References 21

CHAPTER 2 Reading the Problem Like a Judge Does **22**
2.1 The Line Nobody Read 22
2.2 The Anatomy of a Problem Statement 23
2.3 A Reading Checklist 24
2.4 The Three Shapes of Judge Input 25
2.5 ADDREV: Reading “Omit Leading Zeros” 26
2.5.1 A wrong version, and a tool that finds it 28
2.6 FCTRL2: Reading the Size of the Answer 32
2.7 ONP: Reading an Unfamiliar Rule 35
2.8 Using It for Real 40
2.9 Common Mistakes, and How to Avoid Them 41
2.10 Chapter Summary 42
2.11 Exercises 42

2.12 Solutions to the Exercises 43
References 43

CHAPTER 3 Picking the Right Algorithm for the Constraints 45
3.1 The Queue at the Photocopy Shop 45
3.2 From a Time Limit to a Budget of Steps 46
3.3 Counting the Steps of an Algorithm 48
3.4 FCTRL: When a Formula Replaces a Loop 50
3.5 LASTDIG: Halving the Exponent 54
3.6 PRIME1: The Segmented Sieve 57
3.7 Signals in the Statement 61
3.8 Using It for Real 62
3.9 Common Mistakes, and How to Avoid Them 62
3.10 Chapter Summary 63
3.11 Exercises 63
3.12 Solutions to the Exercises 63
References 64

CHAPTER 4 Fast Input and Output 65
4.1 One Trip per Parcel 65
4.2 Where the Time Goes 65
4.2.1 What one `std::cin >> x` really does 67
4.3 Measuring the Ways of Reading 67
4.3.1 The memory cost of reading everything 71
4.4 Measuring the Ways of Writing 72
4.5 The Rules 74
4.6 INTEST: Enormous Input 75
4.7 INOUTEST: Enormous Input and Output 77
4.8 Portability Notes 79
4.9 Using It for Real 79
4.10 Common Mistakes, and How to Avoid Them 80
4.11 Chapter Summary 81
4.12 Exercises 81
4.13 Solutions to the Exercises 82
References 83

CHAPTER 5 Compiler Flags, Pragmas and Portability 84
5.1 A Program That Worked at Home 84
5.2 The Idea in Plain Words 85
5.2.1 From source text to a running program 85
5.2.2 Flags, pragmas, portability 85
5.2.3 Three kinds of difference 85
5.3 The Problem We Carry Through: FCTRL 86
5.3.1 The statement in our own words 86
5.3.2 The idea: count the fives 86
5.3.3 Why the program divides instead of multiplying 87
5.3.4 The program 87
5.4 What the Judge Does With Your Source 89
5.4.1 The compiler on the other side 89
5.4.2 Asking the compiler what it is 89
5.4.3 The sizes of the integer types 91
5.5 Flags That Find Bugs 91
5.5.1 Warnings 91
5.5.2 Sanitizers 92
5.5.3 The optimizer assumes that undefined behavior never happens 93

5.5.4 The debug mode of the standard library ... 94
5.6 Flags That Make Code Faster ... 94
5.6.1 Optimization levels ... 94
5.6.2 What we measured ... 95
5.6.3 Optimization from inside the source ... 97
5.7 The Pragma Risk Ladder ... 98
5.8 A Portability Checklist ... 99
5.8.1 C++ ... 99
5.8.2 Python ... 99
5.8.3 A script that checks before you submit ... 100
5.9 Testing It ... 101
5.10 Using It for Real ... 102
5.11 Common Mistakes, and How to Avoid Them ... 102
5.12 Chapter Summary ... 103
5.13 Exercises ... 103
5.14 Solutions to the Exercises ... 103
References ... 104

CHAPTER 6 Memory and Data-Structure Choices Under a Byte Budget ... 106

6.1 A Suitcase With a Weight Limit ... 106
6.2 The Idea in Plain Words ... 107
6.2.1 Where a program keeps its data ... 107
6.2.2 The weighing formula ... 107
6.2.3 Memory also costs time ... 107
6.3 What One Element Really Costs ... 108
6.3.1 Measuring it in C++ ... 108
6.3.2 Reading the numbers ... 109
6.3.3 Measuring it in Python ... 110
6.4 PRIME1: A Range Too Large to Store ... 112
6.4.1 The problem ... 112
6.4.2 Weighing the obvious solution ... 112
6.4.3 The idea: a small sieve and a window ... 112
6.4.4 A worked example by hand ... 113
6.4.5 The programs ... 113
6.5 LKS: When the Table Does Not Fit ... 116
6.5.1 The problem ... 116
6.5.2 The textbook solution and its weight ... 116
6.5.3 One row is enough ... 117
6.5.4 A worked example by hand ... 118
6.5.5 Python cannot pass LKS ... 119
6.6 The Stack Is Smaller Than You Think ... 119
6.6.1 A graph shaped like a line ... 119
6.6.2 Where the recursion breaks ... 120
6.6.3 The same story in Python ... 121
6.7 Testing It ... 121
6.8 Using It for Real ... 122
6.9 Common Mistakes, and How to Avoid Them ... 123
6.10 Chapter Summary ... 123
6.11 Exercises ... 123
6.12 Solutions to the Exercises ... 124
References ... 125

CHAPTER 7 Classic Implementation Gotchas That Cause Wrong Answer ... 126

7.1 Ten Meters of Fence, Eleven Posts ... 126
7.2 The Idea in Plain Words ... 126

7.3 Overflow: EIGHTS ... 127
7.3.1 The problem ... 127
7.3.2 The idea, with a proof ... 127
7.3.3 How big does it get? ... 128
7.3.4 The same program with `int` ... 129
7.3.5 Testing it ... 130
7.4 Beyond Every Built-in Type: FCTRL2 ... 131
7.4.1 The problem ... 131
7.4.2 How far does `unsigned long long` go? ... 131
7.4.3 Digits in an array, as on paper ... 132
7.4.4 The symptom ... 133
7.4.5 Testing it ... 134
7.5 Zeros That Must Disappear: ADDREV ... 134
7.5.1 The problem ... 134
7.5.2 Where the gotcha lives ... 134
7.5.3 The programs ... 134
7.5.4 Testing it ... 136
7.6 State That Leaks Between Test Cases: CANDY ... 136
7.6.1 The problem ... 136
7.6.2 The idea, with a proof ... 137
7.6.3 The leak ... 137
7.7 Division, Remainders and Floating Point ... 140
7.7.1 Two languages, two answers ... 140
7.7.2 Floating point ... 142
7.8 Off-by-One and the Shape of the Input ... 142
7.8.1 Counting posts ... 142
7.8.2 The input is not always what it seems ... 143
7.8.3 The shape of the output ... 143
7.8.4 A comparator that says “less or equal” ... 143
7.9 Testing It ... 144
7.10 Using It for Real ... 144
7.11 Common Mistakes, and How to Avoid Them ... 145
7.12 Chapter Summary ... 145
7.13 Exercises ... 146
7.14 Solutions to the Exercises ... 146
References ... 147

CHAPTER 8 Debugging TLE and WA: Stress Testing and Brute-Force Checkers ... 148

8.1 Counting the Money Twice ... 148
8.2 The Idea in Plain Words ... 148
8.3 The Harness ... 149
8.3.1 Checking the generator itself ... 151
8.4 INVCNT: Counting Inversions ... 152
8.4.1 The problem ... 152
8.4.2 The brute force ... 152
8.4.3 The fast idea: count while merging ... 152
8.5 Hunting a Wrong Answer ... 156
8.5.1 A program with two slips ... 156
8.5.2 Finding it and shrinking it ... 156
8.5.3 The bug that small tests cannot see ... 157
8.6 Hunting a Time Limit ... 157
8.6.1 Measure, do not guess ... 157
8.6.2 How does the time grow? ... 158
8.6.3 Python on the largest input ... 159
8.7 PALIN: A Wrong Answer and a Hidden Quadratic Step ... 160

8.7.1 The problem 160
8.7.2 The idea 160
8.7.3 The programs 160
8.7.4 A wrong answer: the missing carry 163
8.7.5 A time limit that random tests cannot see 163
8.8 More Tools for Runtime Errors 164
8.8.1 Stop at the line that goes wrong 164
8.8.2 Assertions 165
8.8.3 Print to the error stream 165
8.8.4 Bisecting a file of many test cases 165
8.8.5 When the answer is not unique 165
8.9 Using It for Real 166
8.10 Common Mistakes, and How to Avoid Them 166
8.11 Chapter Summary 167
8.12 Exercises 167
8.13 Solutions to the Exercises 167
References 168

CHAPTER 9 Dynamic Programming Patterns 169
9.1 Bu Wati's Market Days 169
9.2 The Idea in Plain Words 169
9.2.1 Smaller questions with the same shape 169
9.2.2 Working it by hand 170
9.2.3 Five questions for every DP 170
9.3 A First Attempt: Plain Recursion 171
9.4 Pattern 1: A Line of Choices (FARIDA) 173
9.5 Pattern 2: Counting Ways (ACODE) 175
9.6 Pattern 3: Paths in a Grid (BYTESM2) 177
9.7 Pattern 4: Choosing Items Under a Budget (KNAPSACK) 179
9.7.1 One row, and the direction that matters 179
9.7.2 The programs 181
9.8 Pattern 5: Two Strings (EDIST) 183
9.9 Pattern 6: Intervals (MIXTURES) 186
9.10 Testing It: Every Pattern Against a Brute Force 188
9.11 Making It Better: Speed in Both Languages 190
9.12 Using It for Real 191
9.13 Common Mistakes, and How to Avoid Them 192
9.14 Chapter Summary 192
9.15 Exercises 193
9.16 Solutions to the Exercises 193
References 194

CHAPTER 10 Greedy Algorithms and Exchange Arguments 196
10.1 Pak Harun's Community Hall 196
10.2 The Idea in Plain Words 196
10.3 A First Attempt: Change at the Cashier 197
10.4 Picking Activities (BUSYMAN) 199
10.4.1 Which rule? 199
10.4.2 Why earliest end is right 201
10.4.3 The programs 202
10.4.4 When the events have values 203
10.5 Pairing Two Lists (FASHION) 204
10.6 Places for Wishes (BAISED) 206
10.7 Moving Goods Along a Line (GERGOVIA) 208
10.8 Testing It: Every Greedy Against a Brute Force 210

10.9 Making It Better: Speed in Both Languages ... 210
10.9.1 When any two piles may be merged ... 211
10.10 Using It for Real ... 212
10.11 Common Mistakes, and How to Avoid Them ... 213
10.12 Chapter Summary ... 213
10.13 Exercises ... 213
10.14 Solutions to the Exercises ... 214
References ... 216

CHAPTER 11 Binary Search on the Answer and Two Pointers ... 217
11.1 Bu Ratna's Night Market ... 217
11.2 The Idea in Plain Words ... 218
11.2.1 A question that changes its answer once ... 218
11.2.2 The same template in a sorted array ... 218
11.2.3 When no upper end is known ... 219
11.3 A First Attempt: Try Every Value ... 219
11.4 Maximizing the Minimum (AGGRCOW) ... 221
11.5 The Highest Saw (EKO) ... 223
11.6 Binary Search on Real Numbers (PIE) ... 226
11.7 Minimizing the Maximum, and Rebuilding the Answer (BOOKS1) ... 228
11.8 Two Pointers (HOTELS) ... 232
11.8.1 The mirror problem: the shortest run that is enough ... 234
11.9 Testing It: Every Search Against a Brute Force ... 235
11.10 Making It Better: Speed and Library Help ... 235
11.11 Using It for Real ... 236
11.12 Common Mistakes, and How to Avoid Them ... 236
11.13 Chapter Summary ... 237
11.14 Exercises ... 237
11.15 Solutions to the Exercises ... 238
References ... 239

CHAPTER 12 Graph Algorithms Toolkit ... 240
12.1 A Driver, a Timetable and a Chessboard ... 240
12.2 The Idea in Plain Words ... 241
12.2.1 Vertices, edges and the words around them ... 241
12.2.2 Storing a graph: the adjacency list ... 241
12.2.3 Seven questions, seven tools ... 241
12.3 Is It a Tree? Union-Find (PT07Y) ... 242
12.3.1 The problem in our own words ... 242
12.3.2 The idea ... 242
12.3.3 How union-find works ... 243
12.3.4 Running it ... 245
12.4 Breadth-First Search: Knight Moves (NAKANJ) ... 245
12.4.1 The problem in our own words ... 245
12.4.2 The idea: explore in rings ... 246
12.4.3 Running it ... 248
12.5 Two Groups Without Conflict: Bipartite Graphs (BUGLIFE) ... 249
12.5.1 The problem in our own words ... 249
12.5.2 The idea ... 249
12.5.3 Running it ... 252
12.6 The Longest Path in a Tree (PT07Z) ... 253
12.6.1 The problem in our own words ... 253
12.6.2 The idea: two searches ... 253
12.6.3 Running it ... 255
12.7 Shortest Paths with Weights: Dijkstra (SHPATH) ... 255

12.7.1 The problem in our own words 255
12.7.2 The idea: settle the nearest city first 256
12.7.3 A heap with lazy deletion 256
12.7.4 Running it, and the cost of 1,000 queries 259
12.8 Doing Things in Order: Topological Sorting (TOPOSORT) 260
12.8.1 The problem in our own words 260
12.8.2 Kahn's algorithm 260
12.8.3 Running it 263
12.9 Connecting Everything Cheaply: Kruskal (MST) 264
12.9.1 The problem in our own words 264
12.9.2 Kruskal's algorithm and why it is right 264
12.9.3 Running it 266
12.10 Testing the Toolkit 266
12.11 Using It for Real 267
12.12 Common Mistakes, and How to Avoid Them 268
12.13 Chapter Summary 269
12.14 Exercises 269
12.15 Solutions to the Exercises 270
References 271

CHAPTER 13 Number Theory, Geometry and Strings Toolkit 273
13.1 A Field, a Receipt and a Chat Log 273
13.2 The Idea in Plain Words 274
13.2.1 Remainders keep numbers small 274
13.2.2 Exact integers for geometry 274
13.2.3 Never look at the same character twice 274
13.3 Huge Powers: Fast Exponentiation (LASTDIG) 275
13.3.1 The problem in our own words 275
13.3.2 The idea: square and multiply 275
13.3.3 Running it 276
13.4 Counting Coprimes for a Million Numbers: Euler's Totient (ETF) 276
13.4.1 The problem in our own words 276
13.4.2 The product formula 277
13.4.3 Running it 278
13.5 Is a 19-Digit Number Prime? Miller–Rabin (PON) 278
13.5.1 The problem in our own words 278
13.5.2 From Fermat to Miller–Rabin 278
13.5.3 The 128-bit product 279
13.5.4 Running it 281
13.6 A Geometry Kit: Orientation, Convex Hull and Area 282
13.6.1 The cross product 282
13.6.2 The convex hull 282
13.6.3 The area of a polygon 282
13.7 The Closest Pair of Points (CLOPPAIR) 286
13.7.1 The problem in our own words 286
13.7.2 Divide and conquer 286
13.7.3 Running it 290
13.8 Finding a Needle in a Haystack: Knuth–Morris–Pratt (NHAY) 291
13.8.1 The problem in our own words 291
13.8.2 Borders and the prefix function 291
13.8.3 Searching with the prefix function 291
13.8.4 Running it 294
13.9 Repeated Prefixes: the Period of a String (PERIOD) 294
13.9.1 The problem in our own words 294
13.9.2 The idea 294

13.9.3 Running it 296
13.10 Counting Distinct Substrings: the Suffix Array (DISUBSTR) 296
13.10.1 The problem in our own words 296
13.10.2 The idea: every substring is a prefix of a suffix 296
13.10.3 Building the arrays 296
13.10.4 Running it 299
13.11 Testing the Toolkit 299
13.12 Using It for Real 299
13.13 Common Mistakes, and How to Avoid Them 300
13.14 Chapter Summary 301
13.15 Exercises 301
13.16 Solutions to the Exercises 301
References 302

CHAPTER 14 Getting to Accepted: End-to-End Case Studies and Where to Go Next 304

14.1 From a First Submission to a Habit 304
14.2 The Workflow 304
14.3 Case Study 1: Prime Path (PPATH) 305
14.3.1 Step 1: reading 305
14.3.2 Step 2: the budget 306
14.3.3 Step 3: the idea and why it is right 306
14.3.4 Step 4: the brute force 306
14.3.5 Step 5: the fast solution 306
14.3.6 Steps 6 and 7: our own tests, then the stress test 308
14.3.7 Step 8: submitting 309
14.4 Case Study 2: The Longest ABC Path (ABCPATH) 309
14.4.1 Steps 1 to 3 309
14.4.2 Steps 4 and 5 309
14.4.3 Step 7 at work: catching a planted bug 311
14.5 Case Study 3: Ones and Zeros (ONEZERO) 312
14.5.1 Steps 1 and 2 312
14.5.2 Step 3: search over remainders 313
14.5.3 Step 5, and the tempting wrong way 314
14.5.4 Steps 6 and 7: a checker instead of a brute force 316
14.6 When the Verdict Is Not Accepted 317
14.7 Where to Go Next 317
14.7.1 Judges for daily practice 317
14.7.2 Contests in 2026 318
14.7.3 Books to keep on the desk 318
14.7.4 A twelve-week plan 319
14.8 Using It for Real 319
14.9 Common Mistakes, and How to Avoid Them 319
14.10 Chapter Summary 320
14.11 Exercises 320
14.12 Solutions to the Exercises 321
References 322

List of Figures

Figure 1.1 What a judge does with a submission: it compiles the source once, runs the program on every hidden test within the time and memory limits, compares each output with the expected output, and stops at the first failure. 3

Figure 2.1 The six zones of a problem statement and what to write down from each. The constraints, marked in amber, decide both the algorithm and the edge cases. 23

Figure 3.1 Steps needed by the common complexity classes as the input size grows, against about one second of C++ (teal band) and of Python (amber band). Where a curve crosses a band is the largest input that class can handle in a second. 49

Figure 4.1 Measured time to read and to write five million numbers, best of three runs, C++ in teal and Python in amber, the slowest method in red. The absolute times depend on the machine; the ratios are what matter. 71

Figure 5.1 The same 4×108 comparisons under nine flag settings, on a logarithmic scale; red bars use AVX2 instructions. 96

Figure 5.2 The pragma risk ladder: each rung up buys speed on some machines and adds a way to fail on others. 98

Figure 6.1 Bytes per element for 106 values, measured with a counting allocator in C++ and with `tracemalloc` in Python; the scale is logarithmic. 111

Figure 6.2 The segmented sieve: the small primes up to 109 cross out their multiples inside the window $[m, n]$; the numbers outside the window are never stored. 113

Figure 7.1 A 32-bit `int` as a circle: one step past `INT_MAX` lands on `INT_MIN`; the EIGHTS formula for a large k needs 49 bits. 130

Figure 7.2 ADDREV on the pair 4301 and 6: the sum 1040 reversed is "0401", and the zero in front must be dropped before printing. 134

Figure 8.1 The brute force and the merge-sort program on random arrays, on logarithmic scales; the dashed line is INVCNT's time limit. 159

Figure 9.1 One item of size 2 and value 3 added to an all-zero row. Downward, every cell reads a value from before the item, so the item is used at most once; upward, cell 4 reads cell 2 after it already contains the item, so the item is counted twice, and cell 6 three times. 180

Figure 9.2 The edit-distance table for SUNDAY (rows) and SATURDAY (columns); every cell follows from its left, upper and upper-left neighbors by Equation (9.5), and the shaded cells form one cheapest path with distance 3. 184

Figure 10.1 Earliest end first on six activities: the three shaded ones are taken in the numbered order; each skipped activity starts before the person is free again. 201

Figure 10.2 Uncrossing two pairs never lowers the sum of products; here it raises it by $(7 - 2)(9 - 3) = 30$. 204

Figure 11.1 The answers to "can four stalls keep distance d?" form a block of yes and a block of no; the numbered marks are the first four questions of the binary search. 221

Figure 11.2 The two-pointer window on prices 2, 1, 3, 4, 5 with budget 12: the right end advances one step per row, and the left end jumps forward only when the sum exceeds the budget. 233

Figure 12.1 The graph toolkit of this chapter: the question a statement asks, the technique that answers it, and the SPOJ problem used to practice it. 242

Figure 12.2 Our four-city test map with the resulting Dijkstra distances from Surabaya; the highlighted edges form the shortest-path tree. 256

Figure 13.1 The thirteen points of **Listing 13.7**: seven hull vertices, five interior points, and one point on a hull edge that the monotone chain drops. 286

Figure 13.2 The combine step of the closest-pair algorithm: the best pair of each half gives d, and only points in the strip of width $2d$ around the dividing line need to be compared across it. 287

Figure 14.1 The end-to-end workflow used in every case study of this chapter, with the ways back after a failed stress test or a verdict other than Accepted. 305

Figure 14.2 Breadth-first search over remainders modulo 7: the highlighted path appends 0, 0, 1 to the starting 1 and reaches remainder 0, so the answer is 1001. 313

List of Tables

Table 1.1 The verdicts of a judge, how SPOJ names them, what they mean and where to start looking for the bug. 4
Table 1.2 The workbench of this book and the versions that were current when it was written. 6
Table 2.1 A reading checklist: eight questions to answer before writing code, and the verdict that skipping each one tends to produce. 24
Table 2.2 The shunting-yard algorithm on `a+b*c-d` with ONP's priorities. 36
Table 3.1 The complexity budget in C++: the largest input size in the constraints and the slowest complexity that fits in about one second. For Python, move one or two rows down. 49
Table 3.2 Signals in a problem statement and the family of methods they usually point to, with the chapter of this book that treats each family. 61
Table 4.1 The reader `readInt` of **Listing 4.2** on the input `-307`, one character per row. 69
Table 5.1 What happens to a C++ submission on SPOJ, and who controls each step. 85
Table 5.2 Integer sizes in the two 64-bit data models; the bold row is the one that differs. 91
Table 5.3 The output of `ub_check.cpp` under different compilers and optimization levels. 94
Table 5.4 Time of `flags_bench.cpp` under nine settings (fastest of three runs, g++ 13.3, Xeon at 2.1 GHz). 96
Table 5.5 Features that break when a C++ program moves to another platform, and what to write instead. 99
Table 6.1 Rules of thumb for choosing a container by its weight (g++ 13 with libstdc++, CPython 3.11). 111
Table 6.2 The single row `best` of `lks.cpp` after each item, for capacity 10. 118
Table 6.3 Recursive and iterative DFS on a path, under different stack sizes (g++ 13.3, `-O2`). 121
Table 7.1 Seven families of implementation gotchas, the symptom each leaves, and where this chapter treats it. 127
Table 7.2 Integer division, remainder and rounding in C++ and Python. 141
Table 7.3 Stress-test results for the programs of this chapter (300 random tests per run). 144
Table 8.1 What each verdict usually means, and the first tool to reach for. 149
Table 8.2 Time of the two INVCNT programs on random arrays of growing size (fastest of three runs, September 2026). 158
Table 8.3 Time of three PALIN programs on inputs of a million digits. 163
Table 9.1 The brute forces and small inputs used to check the six DP programs. 189
Table 9.2 Largest inputs and measured CPU time; the last column is our judgment of how much room Python has. 190
Table 9.3 Six DP patterns: the state, the last decision that gives the recurrence, and the order of filling. 191
Table 10.1 The brute forces used to check the four greedy programs. 210
Table 10.2 Largest inputs and measured CPU time for the four greedy programs. 211
Table 10.3 Questions that decide between a greedy algorithm and something stronger. 212
Table 11.1 The brute forces used to check the five programs of this chapter. 235
Table 11.2 Largest inputs and measured CPU time for the five programs. 235
Table 11.3 Typical shapes of problems solved by binary search on the answer and by two pointers. 236
Table 12.1 Three ways to store a graph, with the cost of the two questions algorithms ask most. 241
Table 12.2 Time on our largest generated input per problem, rounded from the runs above, and whether a Python submission is realistic. 267
Table 13.1 The problems of this chapter, the limit that rules out the naive approach, and the technique that fits. 274
Table 13.2 The prefix function of `aabaab`, one prefix per row. 291

Table 14.1 The eight steps of the workflow, the questions each one answers, and where this book teaches them. 305
Table 14.2 What each verdict usually means, what to do next, and where this book discusses it. 317
Table 14.3 Where to practice after SPOJ, and what each judge is best for. 318
Table 14.4 A twelve-week practice plan after this book, with one habit to practice deliberately in each block. 319

List of Listings

Listing 1.1 `.vscode/tasks.json`: a build task that compiles the file in the editor with the options of this book. 9
Listing 1.2 `check_env.cpp`: reports the compiler, the C++ standard and the sizes of the integer types. 9
Listing 1.3 `check_env.py`: reports the Python version and shows that Python's integers have no fixed size. 10
Listing 1.4 `test_life.cpp`: the solution of TEST in C++. 11
Listing 1.5 `test_life.py`: the solution of TEST in Python. 12
Listing 1.6 `gen_test_life.py`: writes four tests for TEST, with the expected outputs computed by the simplest possible code. 13
Listing 1.7 `minijudge.py`: compiles a C++ program (or runs a Python one) on every test of a folder, with a time limit, and prints a verdict the way SPOJ does. 13
Listing 1.8 `test_wa.cpp`: a wrong solution that tests for 42 only after printing it. 16
Listing 1.9 `test_re.py`: a wrong solution that keeps the numbers in a list of fixed size and crashes on long inputs. 17
Listing 1.10 `test_ce.cpp`: a solution with a missing semicolon. 17
Listing 2.1 `sum_until_eof.cpp`: the "until end of file" input shape in C++. 25
Listing 2.2 `sum_until_eof.py`: the same input shape in Python, by reading all tokens at once. 26
Listing 2.3 `addrev.cpp`: ADDREV with integer digit reversal. 27
Listing 2.4 `addrev.py`: ADDREV in Python, reversing through strings. 27
Listing 2.5 `addrev_bug.cpp`: a wrong ADDREV that keeps the leading zeros of the reversed sum. 28
Listing 2.6 `stress.py`: runs a generator, a brute force and a program on many seeds, and prints the first input on which the answers differ. 29
Listing 2.7 `gen_addrev.py`: small random ADDREV inputs, rich in zeros. 31
Listing 2.8 `brute_addrev.py`: ADDREV straight from the definition. 31
Listing 2.9 `fctrl2.cpp`: FCTRL2 with schoolbook multiplication on a vector of decimal digits. 32
Listing 2.10 `fctrl2.py`: FCTRL2 in Python, whose integers have no fixed size. 33
Listing 2.11 `check_fctrl2.py`: checks a FCTRL2 program on every possible input. 34
Listing 2.12 `onp.cpp`: ONP with the shunting-yard algorithm; the priorities are those of the statement. 36
Listing 2.13 `onp.py`: the same algorithm in Python, with a list as the stack. 37
Listing 2.14 `brute_onp.py`: ONP by recursive descent, one grammar level per priority. 38
Listing 2.15 `gen_onp.py`: random ONP expressions with minimal and extra brackets. 39
Listing 3.1 `ops_budget.cpp`: measures simple loop steps per second and the time to sort 107 numbers in C++. 46
Listing 3.2 `ops_budget.py`: the same measurement in Python. 47
Listing 3.3 `fctrl_naive.cpp`: FCTRL by counting the factors 5 of every multiple of 5, correct and far too slow. 50
Listing 3.4 `fctrl.cpp`: FCTRL with Legendre's formula, about 13 divisions per case. 51
Listing 3.5 `fctrl.py`: the same formula in Python, reading all input at once. 51

Listing 3.6 `brute_fctrl.py`: FCTRL by computing $n!$ in full and counting its trailing zeros. 52
Listing 3.7 `gen_fctrl.py`: small FCTRL inputs with the boundary values of the powers of 5. 52
Listing 3.8 `make_fctrl_tests.py`: the sample and a maximum-size test for FCTRL, in the format of `minijudge.py`. 53
Listing 3.9 `lastdig.cpp`: LASTDIG with repeated squaring modulo 10. 55
Listing 3.10 `lastdig.py`: the same method in Python; the built-in `pow(a, b, 10)` does the same. 56
Listing 3.11 `prime1.cpp`: PRIME1 with a segmented sieve; the base primes are computed once, each range is sieved separately. 58
Listing 3.12 `prime1.py`: the same segmented sieve in Python, crossing out with slice assignment. 59
Listing 4.1 `gen_intest.py`: writes an INTEST-shaped input with n random numbers. 67
Listing 4.2 `read_bench.cpp`: four ways to read the same input in C++, from plain `cin` to a reader of our own on top of `fread` 67
Listing 4.3 `read_bench.py`: three ways to read the same input in Python. 69
Listing 4.4 `write_bench.cpp`: four ways to write the same output in C++. 72
Listing 4.5 `write_bench.py`: three ways to write the same output in Python. 73
Listing 4.6 `intest.cpp`: INTEST with an `fread`-based reader of non-negative numbers. 75
Listing 4.7 `intest.py`: INTEST in Python, reading all tokens at once. 76
Listing 4.8 `inoutest.cpp`: INOUTEST with unsynchronized streams and the output collected in one string. 78
Listing 4.9 `inoutest.py`: INOUTEST in Python, reading all tokens and joining all products. 78
Listing 5.1 `fctrl.cpp`: the trailing zeros of $N!$, written to behave the same on every platform. 87
Listing 5.2 `fctrl.py`: the same count in Python, reading the whole input at once. 87
Listing 5.3 `platform_probe.cpp`: prints the facts about a compiler and a machine that decide whether a program behaves the same elsewhere. 89
Listing 5.4 `platform_probe.py`: the Python version, implementation and limits that can differ between your machine and the judge. 90
Listing 5.5 `fctrl_mulloop.cpp`: the same formula with a growing power of 5; correct inside the constraints, undefined behavior just outside them. 92
Listing 5.6 `ub_check.cpp`: the loop condition `i > 0` depends on overflow, so the number of steps depends on the optimization level. 93
Listing 5.7 `flags_bench.cpp`: 4×108 comparisons whose speed depends only on the compiler flags 94
Listing 5.8 `flags_bench_pragma.cpp`: the benchmark with an optimize pragma and a target pragma, placed after the includes. 97
Listing 5.9 `portability_check.py`: compiles every program under several standards, compilers and checking modes and reports the failures. 100
Listing 5.10 `brute_fctrl.py`: the brute force for FCTRL builds $N!$ and counts the zeros; only small N are feasible. 101
Listing 6.1 `memsize.cpp`: a counting allocator measures the heap bytes that each container uses for 106 values. 108
Listing 6.2 `memsize.py`: `tracemalloc` measures the bytes per element of Python containers holding 106 values. 110
Listing 6.3 `prime1.cpp`: the segmented sieve; memory depends on n and on $n - m$, never on n. 113
Listing 6.4 `prime1.py`: the segmented sieve on `bytearray` windows, crossing out whole slices at once. 114
Listing 6.5 `lks_table.cpp`: the knapsack with the full table; correct, but the largest tests need about 4 GB. 116
Listing 6.6 `lks.cpp`: the knapsack with a single row of $K + 1$ values, visited from high to low capacity. 117
Listing 6.7 `lks.py`: the one-row knapsack in Python; correct, but far too slow for the largest tests. 119
Listing 6.8 `deep_dfs.cpp`: depth-first search on a path of n vertices, recursive and with an explicit stack. 119
Listing 6.9 `brute_lks.py`: the LKS brute force tries every subset of the items. 122

Listing 7.1 `eights.cpp`: the formula of Equation (7.1), computed in 64-bit arithmetic. 128
Listing 7.2 `eights.py`: the same formula in Python, whose integers never overflow. 128
Listing 7.3 `eights_int.cpp`: the formula with `int`; clean compilation, correct samples, wrong answers for large k. 129
Listing 7.4 `brute_eights.py`: the brute force for EIGHTS searches upward and needs no formula. 130
Listing 7.5 `fctrl2_ull.cpp`: factorials in the largest built-in type; right up to 20! and silently wrong from 21! on. 131
Listing 7.6 `fctrl2.cpp`: factorials of up to 158 digits, stored least significant digit first and multiplied as on paper. 132
Listing 7.7 `fctrl2.py`: FCTRL2 in Python, with `math.factorial`. 133
Listing 7.8 `addrev.cpp`: reversals that always drop leading zeros, and schoolbook addition on digit strings. 134
Listing 7.9 `addrev.py`: ADDREV in Python; `int()` of a reversed string drops the leading zeros. 135
Listing 7.10 `candy.cpp`: the moves of Equation (7.5), with every variable of a block created fresh inside the loop. 137
Listing 7.11 `candy_leak.cpp`: the same program with the total declared outside the loop; every block after the first inherits the candies of the blocks before it. 138
Listing 7.12 `candy.py`: CANDY in Python, reading all tokens at once and stopping at the sentinel -1. 139
Listing 7.13 `arith_demo.cpp`: integer division, remainder, rounding and floating-point comparison in C++. 140
Listing 7.14 `arith_demo.py`: the same questions in Python; several answers differ from C++. 141
Listing 8.1 `stress.py`: runs a generator, a brute force and a candidate on many seeds and stops at the first disagreement. 149
Listing 8.2 `gen_invcnt.py`: random INVCNT inputs with distinct values and the blank lines of the real input format; the largest array size can be chosen. 151
Listing 8.3 `invcnt_brute.cpp`: the brute force looks at every pair; it is obviously right and takes quadratic time. 152
Listing 8.4 `invcnt.cpp`: inversion counting by merge sort, with a 64-bit counter. 153
Listing 8.5 `invcnt.py`: bottom-up merge sort that counts inversions, without recursion. 154
Listing 8.6 `measure.py`: runs a command on an input file and reports the time and the peak memory. 157
Listing 8.7 `palin.cpp`: the next palindrome in linear time on the digit string. 160
Listing 8.8 `palin.py`: the next palindrome in Python, on strings only. 161
Listing 8.9 `brute_palin.py`: the PALIN brute force counts upward until it meets a palindrome. 162
Listing 8.10 `bounds_demo.cpp`: a prefix-sum loop that reads one element past the end of the vector. 164
Listing 9.1 `dp_intro.cpp`: plain recursion, recursion with a memo, and a table, all computing Equation (9.1). 171
Listing 9.2 `farida.cpp`: FARIDA with two rolling values. 173
Listing 9.3 `farida.py`: the same solution in Python, reading all input at once. 174
Listing 9.4 `acode.cpp`: counting readings with Equation (9.2). 175
Listing 9.5 `acode.py`: the same count in Python. 176
Listing 9.6 `bytesm2.cpp`: grid DP with two rows. 177
Listing 9.7 `bytesm2.py`: the same DP in Python, with a padded row instead of border tests. 178
Listing 9.8 `knap_direction.cpp`: the two loop directions against a brute force. 180
Listing 9.9 `knapsack.cpp`: 0/1 knapsack in one row, capacity loop downward. 181
Listing 9.10 `knapsack.py`: the same DP in Python, one new row per item. 182
Listing 9.11 `edist.cpp`: edit distance with two rows. 184
Listing 9.12 `edist.py`: the same DP in Python, with a lean inner loop. 185
Listing 9.13 `mixtures.cpp`: interval DP filled by increasing length. 186
Listing 9.14 `mixtures.py`: the same interval DP in Python, with the colors of all intervals precomputed. 187
Listing 9.15 `tests/stress.py`: the stress-test driver for this chapter's six problems. 189

Listing 10.1 `coin_change.cpp`: greedy change-making against the minimum computed by DP, for two coin systems. 197
Listing 10.2 `greedy_keys.cpp`: three greedy rules for activity selection against a brute force. 199
Listing 10.3 `busyman.cpp`: BUSYMAN, sort by end time and take what fits. 202
Listing 10.4 `busyman.py`: the same greedy in Python. 202
Listing 10.5 `fashion.cpp`: sort both lists and pair them in order. 204
Listing 10.6 `fashion.py`: the same pairing in Python. 205
Listing 10.7 `baised.cpp`: sort the wishes and sum the distances 206
Listing 10.8 `baised.py`: the same greedy with a counting sort. 207
Listing 10.9 `gergovia.cpp`: the sum of the absolute prefix sums. 209
Listing 10.10 `gergovia.py`: the same sum with `itertools.accumulate`. 209
Listing 10.11 `fractional.cpp`: the fractional knapsack by value per kilogram, next to the 0/1 answer found by brute force. 214
Listing 11.1 `bsearch_demo.cpp`: the check for a distance, a linear scan over all distances and a binary search, with the checks counted. 219
Listing 11.2 `aggrcow.cpp`: binary search on the distance with the greedy check. 221
Listing 11.3 `aggrcow.py`: the same search in Python, with an early exit once all cows fit. 222
Listing 11.4 `eko.cpp`: binary search on the saw height, with block input and a 64-bit sum. 224
Listing 11.5 `eko.py`: EKO without binary search, using Equation (11.4) on the sorted heights. 226
Listing 11.6 `pie.cpp`: binary search on a real number with a fixed number of rounds. 227
Listing 11.7 `pie.py`: the same search in Python. 227
Listing 11.8 `books1.cpp`: binary search for the smallest cap, then the split rebuilt from the right. 229
Listing 11.9 `books1.py`: the same search and rebuild in Python. 231
Listing 11.10 `hotels.cpp`: the two-pointer window. 233
Listing 11.11 `hotels.py`: the same window in Python. 233
Listing 11.12 `lis.cpp`: the longest strictly increasing subsequence with `lower_bound`. 238
Listing 12.1 `pt07y.cpp`: a graph is a tree when it has $n - 1$ edges and none of them closes a cycle. 243
Listing 12.2 `pt07y.py`: the same test in Python. 244
Listing 12.3 `nakanj.cpp`: one breadth-first search per query on the 64 squares of the board. 246
Listing 12.4 `nakanj.py`: the same search in Python, with `collections.deque` as the queue 247
Listing 12.5 `buglife.cpp`: BFS two-coloring on every connected component, with a byte-buffer reader for the million-line input. 249
Listing 12.6 `buglife.py`: the same two-coloring in Python. 251
Listing 12.7 `pt07z.cpp`: the diameter of a tree from two breadth-first searches 253
Listing 12.8 `pt07z.py`: the same two searches in Python. 254
Listing 12.9 `shpath.cpp`: Dijkstra with a binary heap and lazy deletion, one search per query, stopping at the target. 257
Listing 12.10 `shpath.py`: the same algorithm with `heapq`. 258
Listing 12.11 `toposort.cpp`: Kahn's algorithm with a min-heap of ready jobs, and cycle detection by counting. 261
Listing 12.12 `toposort.py`: the same algorithm with `heapq` 262
Listing 12.13 `mst.cpp`: Kruskal's algorithm with union-find and a 64-bit total. 264
Listing 12.14 `mst.py`: the same algorithm in Python; Python integers never overflow. 265
Listing 13.1 `lastdig.cpp`: fast exponentiation modulo 10. 275
Listing 13.2 `lastdig.py`: the same loop in Python; the built-in `pow(a, b, 10)` does the same in one call. 275
Listing 13.3 `etf.cpp`: the totient sieve; each query is answered from the table. 277
Listing 13.4 `etf.py`: $\varphi(n)$ by factorization with the primes below 1,000. 277
Listing 13.5 `pon.cpp`: deterministic Miller–Rabin with twelve prime bases and 128-bit products. 279
Listing 13.6 `pon.py`: the same test in Python. 280
Listing 13.7 `geometry.cpp`: orientation by cross product, the monotone-chain convex hull, and the shoelace area. 282

Listing 13.8 `geometry.py`: the same kit in Python. 284
Listing 13.9 `cloppair.cpp`: divide and conquer on squared integer distances; each call leaves its part of the array sorted by y. 287
Listing 13.10 `cloppair.py`: the same algorithm in Python, returning a new list sorted by y from each call. 289
Listing 13.11 `nhay.cpp`: KMP that streams the haystack character by character, storing only the needle and its prefix function. 292
Listing 13.12 `nhay.py`: the same algorithm in Python, which reads each haystack as one string. 293
Listing 13.13 `period.cpp`: the prefix function and Equation (13.9) for every prefix. 294
Listing 13.14 `period.py`: the same in Python. 295
Listing 13.15 `disubstr.cpp`: suffix array by prefix doubling, LCP by Kasai, and Equation (13.10). 297
Listing 13.16 `disubstr.py`: sorted suffixes and neighbor LCPs, enough for $n \leq 1000$. 298
Listing 14.1 `ppath.cpp`: a sieve below 104 and one breadth-first search per test. 306
Listing 14.2 `ppath.py`: the same search in Python, with a generator for the neighbors and a slice assignment in the sieve. 307
Listing 14.3 `abcpath.cpp`: memoized search; the memo table is cleared for every grid. 309
Listing 14.4 `abcpath.py`: the same recurrence evaluated bottom-up, from `Z` down to `A`. 310
Listing 14.5 `onezero_wrong.cpp`: BFS over the numbers themselves, which overflows silently once they pass twenty digits. 314
Listing 14.6 `onezero.cpp`: BFS over the remainders modulo n with parent links, then the digits read back from remainder 0. 315
Listing 14.7 `onezero.py`: the same search in Python, with a list and a head index as the queue. 315

List of Pseudocode

Pseudocode 9.1 Recovering the chosen days from the FARIDA table by walking back from the last day. 194
Pseudocode 10.1 Merging heaps two at a time at the least total cost: always merge the two smallest. 211

List of Equations

(1.1) 4
(1.2) 10
(2.1) 27
(2.2) 32
(3.1) 46
(3.2) 48
(3.3) 48
(3.4) 51
(3.5) 55
(3.6) 57
(3.7) 57
(4.1) 66
(5.1) 86
(5.2) 87
(5.3) 87

(6.1) 107
(6.2) 112
(6.3) 112
(6.4) 113
(6.5) 116
(6.6) 116
(7.1) 127
(7.2) 128
(7.3) 132
(7.4) 132
(7.5) 137
(7.6) 142
(7.7) 142
(7.8) 142
(8.1) 152
(8.2) 153
(8.3) 153
(8.4) 159
(9.1) 170
(9.2) 175
(9.3) 177
(9.4) 179
(9.5) 183
(9.6) 186
(10.1)....201
(10.2)....203
(10.3)....204
(10.4)....206
(10.5)....208
(11.1)....218
(11.2)....218
(11.3)....224
(11.4)....225
(11.5)....226
(11.6)....229
(11.7)....232
(12.1)....256
(12.2)....257
(13.1)....274
(13.2)....275
(13.3)....277
(13.4)....278
(13.5)....279
(13.6)....282
(13.7)....282
(13.8)....291
(13.9)....294
(13.10)....296
(14.1)....306
(14.2)....309
(14.3)....313

Preface

A student submits her first solution to an online judge at midnight and is told, in red, that it is wrong. The program printed the right answers on her laptop. It printed them for every example in the statement. What the judge saw, and she did not, was a line in the statement she had skimmed, an input ten thousand times larger than the sample, a number that no longer fitted in an `int`, and a compiler that was not the one on her desk. Competitive programming is the craft of closing that gap: of writing, in a limited time, a program that is correct for every input the problem allows and fast enough for the largest of them, and of knowing that it is before a judge says so.

SPOJ, the Sphere Online Judge, is one of the oldest and broadest places to learn that craft. It has thousands of problems collected over two decades, accepts programs in dozens of languages, and answers every submission in seconds with a verdict: *Accepted*, *Wrong answer*, *Time limit exceeded*, *Runtime error* or *Compilation error*. This book uses its classic problems — TEST, PRIME1, ADDREV, FCTRL2, AGGRCOW, SHPATH and many more — as the exercises through which the whole craft is taught, from reading a statement to getting a hard problem accepted.

Who this book is for

The book is self-contained, and you are assumed to be a beginner. If you have never taken part in a contest, never submitted a program to an online judge and are not sure what "O(n log n)" means, you are exactly the reader this book was written for. Every term is introduced in plain words before it is used, every idea is first worked by hand on a small example, the mathematics it needs is explained where it is needed, and every program in the book was run before its output was printed here. Chapter 1 begins with a red verdict at midnight and ends with a workbench, a first accepted problem and a small judge of your own.

Readers who already program, and students preparing for olympiads, national contests or the ICPC, will find the other half of the book: every algorithm argued correct and its cost counted, the judge's time and memory budgets turned into concrete limits on what a program may do, the traps that turn a right idea into a wrong answer catalogued one by one, and complete, tested programs ready to be adapted. That principle makes the book long. A shorter book would assume you already knew the hard parts, and such a book is no use to the person who does not.

One idea, two languages

Every problem in this book is solved in **C++17** first, because C++ is the language of competitive programming: it is fast, its standard library holds most of the data structures a contest needs, and every judge accepts it. Where Python can pass the time limit, the same solution is also written in **Python 3**, and the two versions read the same input and print identical output. Python is short and easy to read, and seeing one idea written both ways is the quickest route to understanding the idea rather than the habits of one language. Where Python cannot pass, even with the fastest input and output techniques, the book says so, and explains why.

No program in this book is trusted because it passes the samples. Every one of them is **stress-tested against a brute force**: a second program, deliberately simple and slow, whose correctness is obvious, is run against it on thousands of small random inputs made by a generator, and the

test stops at the first disagreement. The programs are our own teaching programs, written for this book; every statement is retold in our own words, with its SPOJ code and address so that you can read the original. All of it is written and run in **Visual Studio Code**, with the GNU C++ compiler and Python 3.14, on Windows, macOS and Linux alike; Chapter 1 installs them step by step on all three systems.

What is in it

Chapter 1 explains what competitive programming is, what SPOJ offers and what the judge actually does with a submitted program. It installs the workbench, solves the first SPOJ problem, TEST, and builds a small judge of your own that compiles, runs and compares programs the way SPOJ does.

Chapters 2 to 4 are about the problem before the algorithm. Chapter 2 teaches you to read a statement like a judge does: its anatomy, a reading checklist, the three shapes of judge input, and the phrases that decide everything, in ADDREV, FCTRL2 and ONP. Chapter 3 turns a time limit into a budget of steps, counts the steps of an algorithm, and shows how the constraints point to the method, with FCTRL, LASTDIG and the segmented sieve of PRIME1. Chapter 4 measures every way of reading and writing data in both languages and draws the rules for enormous input and output, in INTEST and INOUTEST.

Chapters 5 to 8 are about everything that stands between a right idea and an accepted program. Chapter 5 explains what the judge does with your source: compiler flags that find bugs, flags and pragmas that make code faster, a risk ladder for pragmas, and a portability checklist. Chapter 6 spends the second budget, memory: what one element really costs, a range too large to store in PRIME1, a table that does not fit in LKS, and a stack that is smaller than you think. Chapter 7 catalogues the classic implementation gotchas that cause *Wrong answer* — overflow, numbers beyond every built-in type, zeros that must disappear, state that leaks between test cases, division and floating point, and off-by-one errors. Chapter 8 builds the harness that the rest of the book relies on: a generator, a brute force and a stress test that hunt down wrong answers and hidden slow steps, in INVCNT and PALIN.

Chapters 9 to 11 are the algorithmic core. Chapter 9 teaches dynamic programming through six patterns — a line of choices, counting ways, paths in a grid, a knapsack, two strings and intervals — in FARIDA, ACODE, BYTESM2, KNAPSACK, EDIST and MIXTURES. Chapter 10 shows when a greedy choice is safe and how an exchange argument proves it, in BUSYMAN, FASHION, BAISED and GERGOVIA. Chapter 11 searches for the answer instead of computing it, with binary search on the answer and two pointers, in AGGRCOW, EKO, PIE, BOOKS1 and HOTELS.

Chapters 12 and 13 are toolkits. Chapter 12 assembles the graph algorithms a contest needs most: union-find, breadth-first search, bipartite checking, the longest path in a tree, Dijkstra's shortest paths, topological sorting and Kruskal's minimum spanning tree. Chapter 13 does the same for number theory, geometry and strings: fast exponentiation, Euler's totient, the Miller–Rabin primality test, orientation, the convex hull and polygon area, the closest pair of points, Knuth–Morris–Pratt, the period of a string and the suffix array.

Chapter 14 puts the whole craft together. It turns the chapters into one workflow and follows it end to end on three case studies, PPATH, ABCPATH and ONEZERO, shows what to do when the verdict is not *Accepted*, and suggests where to go next.

Every chapter follows the same path: a story and the problem it poses, the idea in plain words with a small example worked by hand, real SPOJ problems retold in our own words, the idea argued correct and its cost counted, reference programs in C++17 and Python, running them, testing them against a brute force, making them better, using them for real, common mistakes, a summary, exercises with complete solutions, and references.

Every program in this book was run before its output was printed. The programs are collected, chapter by chapter, in the accompanying code folder, from `Chapter01` to `Chapter14`, each with a `cpp` folder, a `python` folder, a `tests` folder holding the generators, brute forces and stress scripts, and a README that explains how to build and run them in a terminal and in VS Code on Windows, macOS and Linux.

How to read it

With VS Code open and a judge account ready. Chapters 1 to 4 come first for everyone, because they build the workbench, the reading habits and the time budget that every later chapter relies on, and Chapter 8's stress-testing harness is worth learning before the algorithmic chapters; after that the chapters climb from easy to hard, and each one names the earlier chapters it builds on. Read each statement and try the problem yourself before you read the solution, even if only for twenty minutes: the struggle is what makes the idea stick. Work the small examples by hand, type the programs in rather than copying them, and when a program works, break it on purpose — make an array one element too small, use an `int` where a `long long` is needed — and watch the stress test catch it. The exercises at the end of each chapter all have complete solutions — use them after your own attempt, not instead of it.

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

About the author

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 engineers meet the same budgets of time and memory that this book teaches. Away from the keyboard he tends the family fruit and durian orchard, where he learned that good software, like a good orchard, is grown patiently, one well-tended piece at a time. The storyteller in him is the reason every chapter of this book begins with a person and a problem rather than with a definition.

Acknowledgments

To the students who worked through the earlier forms of this material, and whose rejected submissions, asked about late at night, showed where an explanation was still thin, and to the colleagues at ITS with whom algorithms and programming have been taught and refined over the years. To the problem setters and maintainers of SPOJ, whose problems have trained a generation of programmers, and to the wider competitive-programming community, whose editorials and

discussions keep the craft alive. The book owes its standard, too, to the official documentation of C++, GCC, Python and VS Code, and to the classic textbooks and research literature on algorithms, which are cited wherever they are used.

Surabaya, September 2026



Irfan Subakti · yifana@gmail.com

Profitina

profitina.com

Chapter 1

Welcome to Competitive Programming and SPOJ

A program that works on your laptop is not yet a program that works. An online judge takes your source file, compiles it with a compiler you did not choose, runs it on inputs you have never seen, gives it a fixed number of seconds and megabytes, and compares every character it prints with an answer that was decided long before you started typing. This chapter explains that machine and gets you ready for it. By the end of the chapter you will be able to say what competitive programming is and what SPOJ offers; describe step by step what a judge does with a submission and what each verdict (AC, WA, TLE, RE, CE) tells you about your bug; install a complete C++17 and Python 3.14 workbench, with g++, VS Code and its extensions, on Windows, macOS or Linux; solve and run your first SPOJ problem in both languages; and build a small judge of your own that gives your programs the same verdicts on your own computer, before SPOJ ever sees them.

1.1 A Red Verdict at Midnight

Rina is in her second year of informatics at a university in Surabaya. Her campus team wants to enter a national programming contest, and the senior who coaches the team has given every newcomer the same homework: open an account on SPOJ, the Sphere Online Judge, and solve the problem with the code TEST before the next meeting. The problem looks almost insultingly simple. The input is a list of small numbers, one per line; print them back, one per line, and stop as soon as you meet the number 42.

Rina writes ten lines of C++ in a few minutes. She types the sample input into her program, the sample output appears on the screen exactly as printed on the problem page, and she submits. A few seconds later the status page shows her submission in red: *wrong answer*. She reads her code again. It prints the numbers; it stops at 42. She types the sample again, and again the output is right. At midnight she sends her coach a message: “The judge is broken.”

The judge is not broken, of course. Rina’s program printed the 42 as well, before it stopped, and the sample on the problem page happens to end in a way that hid the difference from her tired eyes. Section 1.12 builds a small judge that finds this mistake in a fraction of a second and tells you on which line it happened. But the more important lesson is the one the coach wanted her to learn on the first day: the judge does not care that the program *almost* works, does not read the code, and does not listen to explanations. It runs the program and compares what it prints. That strictness is exactly what makes a judge useful.

Who gains from learning to satisfy such a strict examiner? Rina and her teammates, obviously, who want to take part in the International Collegiate Programming Contest (ICPC), the largest programming contest for university students, whose regional rounds lead to a world final [1]. High-school students who prepare for the informatics olympiad. Anyone who applies for a software job, because the technical interviews of many companies are small competitive-programming problems solved in front of an interviewer. And, less obviously, every working programmer: the habits that a judge forces on you, reading a specification to the last word, asking

how large the input can get, testing the smallest and the largest case, and checking a fast program against a slow but obviously correct one, are the habits that separate software that usually works from software that works.

Think of a driving test. The examiner does not care how well you drove yesterday or how carefully you meant to check the mirror; they watch what you do on the route they chose, which you did not see beforehand, and they have a checklist. A judge is an examiner with a checklist and a stopwatch, and the route is hidden. This book teaches you to drive so well that the route does not matter.

1.2 What Competitive Programming Is, and What SPOJ Offers

> **DEFINITION.** *Competitive programming* is solving precisely specified algorithmic problems by writing programs that a machine, the *judge*, checks automatically for correct output, running time and memory use. A *problem* consists of a statement, hidden test inputs, the expected outputs for them, a time limit and a memory limit. A *submission* is one source file sent to the judge; the judge's response to it is a *verdict*.

Competitive programming comes in two forms. In a *contest*, many people or teams solve the same set of problems at the same time, typically five to twelve problems in two to five hours, and a scoreboard ranks them by the number of problems solved and the time needed. The ICPC is a contest for teams of three students sharing one computer [1]; the olympiads in informatics are contests for individual school students; and many websites hold contests online every week. In an *archive*, the problems stay online permanently and you solve them whenever you like, at your own pace, as training. SPOJ is above all an archive.

SPOJ, the Sphere Online Judge, is one of the oldest public judges on the web. It holds many thousands of problems, written by volunteers from all over the world, and every one of them has a short code of up to eight capital letters: TEST, PRIME1, ADDREV, FCTRL2 [2]. The code is also the address of the problem: the problem TEST lives at `https://www.spoj.com/problems/TEST/` [3]. SPOJ groups its problems into sections. The *classical* section, the one this book uses, contains problems whose answer is either fully correct or not, with a binary verdict. The *challenge* section scores how good an answer is, for example how short the program is or how close a heuristic comes to an optimum. The *tutorial* section holds easier problems meant for practice. Each classical problem also shows how many people have solved it, which is a useful rough measure of its difficulty: TEST has been solved by a very large number of users, while some problems have been solved by only a handful.

What makes SPOJ especially good for learning is its breadth of languages and its age. It accepts programs in dozens of programming languages, and its problems have accumulated over two decades, so almost every classical technique of algorithm design appears in a problem somewhere, usually in several variants of rising difficulty. Its age also has a price that this book will keep in mind: the judge's compilers and machines are not the newest. At the time of writing, the C++ compiler that most SPOJ problems accept is GCC 8.3, offered as "C++14 (gcc 8.3)" [4], a version released in 2019, and SPOJ's Python 3 is likewise years older than the Python 3.14 we install in this chapter. Section 1.4 explains how the programs of this book deal with that.

Competitive programming is not the same thing as software engineering. A contest problem is small, its specification is complete and exact, it never changes after you start, and its only user is the judge. Real software has vague requirements that change every month, many users, and code that lives for years. Yet the two meet at an important point: a large part of what makes real

software slow or wrong is exactly what competitive programming trains, namely choosing the right algorithm and data structure for the size of the input, and handling the edge cases that a quick test does not cover. A programmer who has solved a few hundred judge problems reads a requirement differently, and asks earlier "what happens when the list is empty?" and "how many records can there be?".

1.3 What the Judge Does With Your Program

When you press the submit button, a chain of steps begins that is the same on every judge in the world, and **Figure 1.1** draws it.

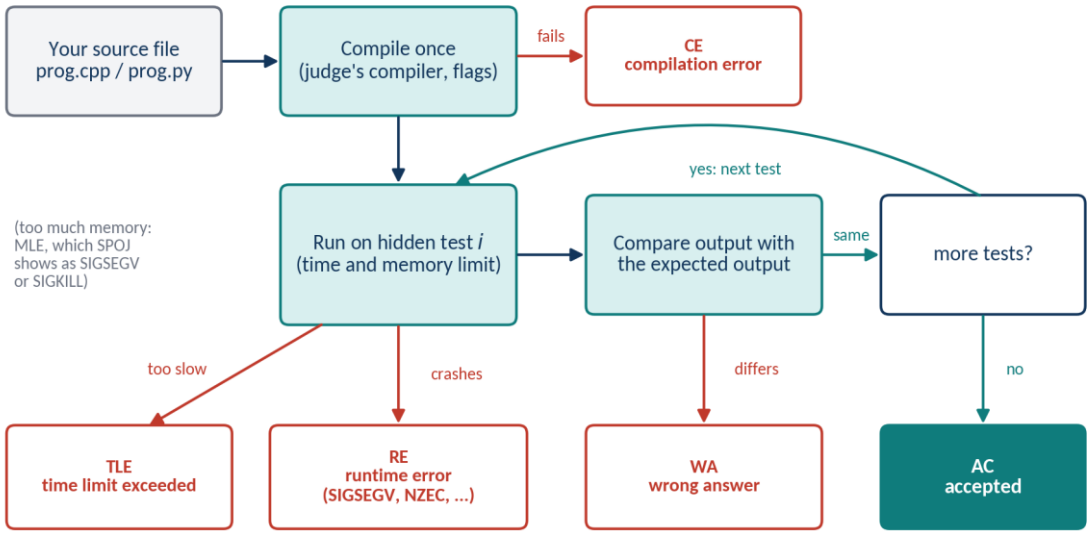


Figure 1.1 What a judge does with a submission: it compiles the source once, runs the program on every hidden test within the time and memory limits, compares each output with the expected output, and stops at the first failure.

Step 1: compile. For a compiled language such as C++, the judge first compiles the source file with its own compiler and its own options. If the compiler refuses the file, the verdict is *compilation error* (CE) and nothing is run. For Python, there is no separate compiling step on the judge; a syntax error shows up when the program starts and is usually reported as a runtime error.

Step 2: run on each hidden test. The problem setter prepared a set of test files, sometimes one large file containing many test cases, sometimes a dozen separate files. The judge starts your program once per test file, connects the file to the program's *standard input* (the stream that `std::cin` and `input()` read, which on your computer is normally the keyboard) and catches everything the program writes to *standard output* (the stream of `std::cout` and `print()`, normally the screen). Your program never opens a file by name; it simply reads and writes, and the judge has put the right things at both ends. While the program runs, the judge measures its time and memory. If the time limit passes before the program ends, the judge kills it: *time limit exceeded* (TLE). If the program ends abnormally, for example because it read outside an array, divided by zero or raised an uncaught exception, the verdict is *runtime error* (RE), and SPOJ adds the name of the signal or reason, such as `SIGSEGV` (a memory access violation), `SIGFPE` (an arithmetic fault such as division by zero), `SIGABRT` (the program aborted itself) or `NZEC` (non-zero exit code, typical for an uncaught Python exception).

Step 3: compare. If the program ended normally within the limits, the judge compares its output with the expected output. Most classical problems use a comparison that ignores differences in whitespace, that is, extra spaces and line breaks, but not in anything else; a few problems accept several correct answers and use a special checker program instead. If the output differs, the verdict is *wrong answer* (WA).

Step 4: stop at the first failure, or accept. The judge moves to the next test only if the current one passed. The first failure decides the verdict; only a program that passes every test gets *accepted* (AC). **Table 1.1** lists the verdicts with the first thing to check for each.

Table 1.1 The verdicts of a judge, how SPOJ names them, what they mean and where to start looking for the bug.

Verdict	SPOJ shows	What happened	Look first at
AC	accepted	Every test passed within the limits	Nothing: move on, but read Section 1.13
WA	wrong answer	The output differs on some test	Output format, edge cases, overflow
TLE	time limit exceeded	The program did not finish in time	The algorithm's complexity (Chapter 3), slow input and output (Chapter 4), an endless loop
RE	runtime error (SIGSEGV, SIGFPE, SIGABRT, NZEC, …)	The program crashed	Array bounds, division by zero, recursion depth, reading past the end of input
CE	compilation error	The compiler refused the source	A language feature the judge's compiler lacks, a missing header, the wrong language chosen
MLE	usually appears as a runtime error	The program used too much memory	Array sizes versus the memory limit (Chapter 6)

KEY INSIGHT. A verdict is a clue, not a grade. WA says the algorithm finished and printed something wrong, so the search is for a wrong idea or an unhandled case; TLE says nothing at all about correctness, only that the program was too slow on some test; RE says the program broke, which is often a bug in indexing rather than in the idea. Reading the verdict this way turns "it failed" into a narrow search.

Two properties of this process deserve to be said aloud, because beginners are surprised by both. First, the hidden tests are not a random sample. A careful problem setter writes tests that hunt for bugs: the smallest legal input, the largest legal input, inputs full of equal values, inputs whose answer is zero, inputs designed so that a program with the wrong complexity runs for minutes. The sample on the problem page exists only to show the format. Passing it proves that you read the format correctly and almost nothing else. Second, the time limit applies to the judge's machine, not to yours. A program that needs 0.9 seconds on your new laptop may need 2 seconds on an older judge machine. Chapter 3 shows how to estimate running time before you write the program, which is the only reliable defense.

The time a program needs is, roughly, the number of simple steps it performs divided by the number of steps the machine can do per second:

$$t \approx \frac{\text{number of steps}}{\text{steps per second}} \tag{1.1}$$

A modern computer running compiled C++ does somewhere between 10^8 and 10^9 simple steps per second, and Python roughly fifty to a hundred times fewer. Equation (1.1) looks too simple to be useful, but it decides a surprising number of problems at a glance, and Chapter 3 is built on it.

1.4 How This Book Works

Every chapter of this book is organized around real SPOJ problems. For each one we do the same seven things, in the same order, because this order is how experienced competitors actually work.

1. **Restate the problem in our own words.** We never copy a statement; we tell you what it asks, and we give its SPOJ code and address so that you can read the original. Retelling a problem in your own words is itself a skill, and Chapter 2 is devoted to it.
2. **Read the constraints** and let them tell us which algorithms are fast enough, with Equation (1.1).
3. **Find the idea** in plain words, usually by working a small example by hand.
4. **Argue that it is correct**, with a proof as short as the idea allows, and **count its cost**.
5. **Write a reference program in C++17**, the main language of the book, and **in Python 3**. The two print exactly the same output. When Python is too slow for a problem even with the fastest techniques, we say so, and why.
6. **Run it** on the sample and on our own inputs, and show the real output.
7. **Test it** against a *brute force*: a second program, deliberately simple and slow, whose correctness is obvious. A *generator* makes many small random inputs, and a *stress test* runs both programs on each of them and stops at the first disagreement. From Chapter 2 on, every problem comes with these three tools.

The programs are our own teaching programs, written for this book and tested on our own inputs. They are not copies of anybody's accepted submission, and they are written to be read: clear names, one idea per function, and comments on the lines that are not obvious. They have not been submitted to SPOJ for you; submitting them would teach you nothing, while typing them, running them and then changing them teaches a great deal.

All code lives in the folder `SPOJ/Code_new`, with one folder per chapter: `Chapter01/cpp` holds the C++ programs, `Chapter01/python` the Python programs, and `Chapter01/tests` the generators, brute forces and test files. Each chapter folder has a `README.md` that lists its programs and repeats the commands to build and run them on each operating system.

About versions. We compile every C++ program with `g++ -std=c++17 -O2 -Wall -Wextra`, with no warnings, and we also check that each one is clean under the *sanitizers* `-fsanitize=address,undefined`, which make the program stop with a clear message at the first out-of-bounds access or undefined operation instead of silently continuing. Because SPOJ's C++ compiler is older, the programs deliberately use only features that GCC 8.3 understands, and every one of them also compiles with `-std=c++14`. The Python solutions are written for Python 3.14 but use nothing newer than f-strings, so they run on the older Python 3 of the judge as well; the local test tools need Python 3.11 or newer. The runs shown in this book were made on a Linux machine with GCC 13.3 and Python 3.14.7.

1.5 Your Workbench: What We Install and Why

We need four things: a C++ compiler, Python, an editor, and two extensions that teach the editor about C++ and Python. **Table 1.2** lists them with the versions that are current in September 2026.

Table 1.2 The workbench of this book and the versions that were current when it was written.

Tool	What it is	Version in September 2026
GCC (`g++`)	The GNU C++ compiler	16.2, released August 7, 2026 [5]; Ubuntu 26.04 ships GCC 15 [6]
MSYS2 (Windows)	A package collection that brings GCC to Windows	package `mingw-w64-ucrt-x86_64-gcc` 16.2.0 [7]
Command Line Tools (macOS)	Apple's compiler Clang, which also answers to `g++`	installed with `xcode-select --install` [8]
Python	The Python interpreter and its standard library	3.14.7, released August 5, 2026 [9]
Visual Studio Code	A free editor for Windows, macOS and Linux	1.139, released September 23, 2026 [10]
C/C++ extension	IntelliSense, building and debugging C++ in VS Code	`ms-vscode.cpptools` [11]
Python extension	Running, checking and debugging Python in VS Code	`ms-python.python` [12]

Why g++? Because it is the compiler of the judge. SPOJ, like almost every judge, compiles C++ with GCC, and GCC has a few habits that other compilers do not share, such as the header `<bits/stdc++.h>` that includes the whole standard library at once. Programs that compile with GCC on your machine compile with GCC on the judge, provided that you avoid features newer than the judge's version. On Windows we take GCC from MSYS2, which is also the route the VS Code documentation recommends [13]; on Linux it comes from the system's package manager; on macOS the system compiler is Clang, which is fine for this book, and Section 1.7 shows how to get the real GCC as well. Any GCC from version 8 onward compiles every program in this book; the newest is simply the best one to install today.

Why C++17 and not something newer? The C++ language gets a new standard every three years: C++17, C++20, C++23. A newer standard offers more conveniences, but a program that uses a C++20 feature fails with a compilation error on a judge whose compiler predates it. C++17 is recent enough to be pleasant and old enough to be safe, and by avoiding the few C++17 additions that GCC 8 lacks we stay safe even on SPOJ. Recent compilers use a newer standard by default, which is why every command in this book passes `-std=c++17` explicitly.

Why Python 3.14? Python 3.14 is the current series, and its seventh maintenance release, 3.14.7, appeared on August 5, 2026 [9]. Python is an excellent second language for competitive programming: its integers never overflow, its strings and lists are convenient, and a correct brute force takes a few lines. Its weakness is speed, and this book is honest about it: for some problems, a Python program with the right algorithm is still too slow for the judge, and we will measure how much slower it is.

Why VS Code? Because it is free, runs on all three systems, handles both languages, and has a built-in terminal. Any editor works; the instructions use VS Code so that one set of menu names fits everyone.

1.6 Installing on Windows 10 and 11

1.6.1 The C++ compiler, from MSYS2

Windows has no C++ compiler that works from a terminal out of the box, so we install GCC from MSYS2, a well-maintained collection of the tools that Linux programmers use [14].

1. Download the installer from `https://www.msys2.org` and run it. Accept the default folder `C:\msys64`; a path without spaces avoids trouble later.
2. From the Start menu open **MSYS2 UCRT64**. The name matters: UCRT64 is the modern variant, and a compiler installed from another MSYS2 terminal does not match it. Type the following command, press Enter to accept all members of the group, and answer `Y`:

   ```
   $ pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain
   ```
3. Add the folder `C:\msys64\ucrt64\bin` to the `Path`, the list of folders in which Windows looks for commands: press the Windows key, type *environment*, open **Edit the system environment variables**, click **Environment Variables**, select **Path** under *User variables*, click **Edit**, then **New**, paste the folder and confirm every dialog with **OK**.
4. Close all terminals, open a new **PowerShell** (a window opened before the change does not see it) and check:

   ```
   PS C:\Users\you> g++ --version
   g++.exe (Rev..., Built by MSYS2 project) 16.2.0
   ```

The first line should name MSYS2 and a version of 13 or newer. If PowerShell answers that `g++` "is not recognized", the `Path` entry is missing or misspelled, or the terminal is older than the change.

An alternative, if you prefer a real Linux environment inside Windows, is the Windows Subsystem for Linux: `wsl --install` in an administrator PowerShell installs Ubuntu, and inside it you follow Section 1.8. Both routes work for the whole book.

1.6.2 Python

Python on Windows is installed with the **Python install manager**, a small program that downloads and switches between Python versions [15]. Install it from the Microsoft Store (search for "Python Install Manager") or from `https://www.python.org/downloads/windows/`, then open a new PowerShell and type:

```
PS C:\Users\you> py install 3.14
PS C:\Users\you> python --version
Python 3.14.7
```

After this, `python` and `py` both start Python 3.14. If `python --version` opens the Microsoft Store instead, close every terminal, open a new one and try again. Everywhere this book writes `python3`, Windows users type `python` or `py`.

1.6.3 VS Code

Install VS Code with `winget install Microsoft.VisualStudioCode` or with the **User Installer** from `https://code.visualstudio.com`. Tick **Add to PATH**, which lets you type `code .` in a terminal to open the current folder in the editor.

1.7 Installing on macOS

1.7.1 The C++ compiler

Open **Terminal** (in **Applications › Utilities**) and type:

```
$ xcode-select --install
```

A dialog offers to install the **Command Line Tools**; accept it. The download is a few hundred megabytes and brings Apple's compiler Clang together with `git`, `make` and the rest of the classic tools [8]. You do not need the full Xcode application. Afterwards:

```
$ g++ --version
Apple clang version 17.x.x (clang-17xx.x.x.x)
```

The exact numbers depend on your macOS version. Notice what the first line says: on a Mac, the command `g++` is not GCC but Apple's Clang under another name. For almost everything in this book that makes no difference, because Clang understands standard C++17 just as well. It makes a difference in one place: Clang does not have GCC's private header `<bits/stdc++.h>`, so a program that includes it compiles on the judge and fails on a Mac. This book includes the standard headers one by one (`<iostream>`, `<vector>`, `<string>`), which works everywhere. If you want the real GCC, the one the judge uses, install **Homebrew** from `https://brew.sh` and then `brew install gcc`; the compiler is then called `g++-16`, with the version number in the name, so that it does not collide with Apple's `g++` [16].

1.7.2 Python and VS Code

macOS does not include a current Python. Download the **macOS installer** for Python 3.14.7 from `https://www.python.org/downloads/`, run it, and at the end double-click **Install Certificates.command** in the folder that opens. Then `python3 --version` prints `Python 3.14.7`. (With Homebrew, `brew install python@3.14` works as well.) For VS Code, download the macOS version from `https://code.visualstudio.com`, drag it into **Applications**, start it, open the Command Palette with **Cmd+Shift+P** and run **Shell Command: Install 'code' command in PATH**.

1.8 Installing on Linux

On Linux everything comes from the distribution's package manager. On Ubuntu, Debian and their relatives:

```
$ sudo apt update
$ sudo apt install build-essential gdb python3 python3-venv
$ g++ --version
g++ (Ubuntu 15.2.0-...) 15.2.0
$ python3 --version
Python 3.14.x
```

The package `build-essential` brings `g++`, `make` and the C library headers; `gdb` is the debugger. Ubuntu 26.04 LTS ships GCC 15 as its default compiler [6] and Python 3.14 as its `python3`; the older Ubuntu 24.04 ships GCC 13.3 and Python 3.12, and both work for the whole book, so there is no need to replace them. On Fedora the equivalent is `sudo dnf install gcc-c++ gdb python3`. Never remove or replace the system Python on Linux; parts of the operating system depend on it.

For VS Code, download the `.deb` package (Ubuntu, Debian, Mint) or the `.rpm` package (Fedora, openSUSE) from `https://code.visualstudio.com` and install it with `sudo apt install ./code_*.deb` or `sudo dnf install ./code-*.rpm`.

1.9 VS Code: Two Extensions and a Build Task

Start VS Code, click the **Extensions** icon in the bar on the left (or press **Ctrl+Shift+X**, on a Mac **Cmd+Shift+X**) and install two extensions published by Microsoft:

- **C/C++** (identifier `ms-vscode.cpptools`). It completes names and marks errors while you type (IntelliSense), and runs the debugger [11]. If the editor underlines correct code in red, open the Command Palette (**Ctrl+Shift+P**), run **C/C++: Select IntelliSense Configuration** and choose the `g++` you installed.

- **Python** (identifier `ms-python.python`). It brings the Pylance checker and the Python debugger [12]. Click the Python version in the status bar at the bottom to choose which interpreter VS Code uses; choose 3.14.

Always open a *folder* (**File › Open Folder**), not a single file: for this chapter, open `Chapter01`. The integrated terminal (**Terminal › New Terminal**) then starts in that folder.

VS Code does not know by itself with which options to compile C++, so we tell it once, in a small file `.vscode/tasks.json` inside the chapter folder. **Listing 1.1** is the file, and the same file is in the folder of every chapter.

Listing 1.1 `.vscode/tasks.json`: a build task that compiles the file in the editor with the options of this book.

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "C++17: build active file",
      "type": "shell",
      "command": "g++",
      "args": ["-std=c++17", "-O2", "-Wall", "-Wextra",
               "${file}", "-o", "${fileDirname}/${fileBasenameNoExtension}"],
      "group": { "kind": "build", "isDefault": true },
      "problemMatcher": ["$gcc"]
    }
  ]
}
```

The task runs `g++` on the file shown in the editor (`${file}`) and writes the program next to it, under the same name without `.cpp` (on Windows, g++ adds `.exe` by itself). Because the task is marked as the default build task, **Ctrl+Shift+B** (Mac: **Cmd+Shift+B**) runs it, and the `$gcc` problem matcher turns every compiler error into a red squiggle at the right line. On a Mac with Homebrew's GCC, change `"g++"` to `"g++-16"`.

Competitive programs read standard input, so it is worth learning how to feed them a file instead of typing. In the terminals of Linux and macOS, and in the classic Windows **Command Prompt**, the operator `<` connects a file to standard input and `>` sends standard output to a file:

```
$ ./test_life < sample.in > my.out
```

PowerShell, the default terminal of VS Code on Windows, does not accept `<`. There you write `Get-Content sample.in | .\test_life.exe`, or you switch the terminal to Command Prompt with the small **v** next to the **+** in the terminal panel. When you type input by hand instead, end it with **Ctrl+D** on Linux and macOS and with **Ctrl+Z** followed by Enter on Windows; that key sends *end of file*, which tells the program that no more input will come.

1.10 Your First Programs: Checking the Workbench

The first two programs do nothing but report on the tools that built and run them, which is the quickest way to confirm that the installation worked and to see a few facts that matter later. **Listing 1.2** is the C++ version.

Listing 1.2 `check_env.cpp`: reports the compiler, the C++ standard and the sizes of the integer types.

```cpp
// check_env.cpp: shows which compiler and C++ standard built this program.
#include <climits>
#include <cstdio>

int main() {
#if defined(__clang__)
    std::printf("compiler: clang %d.%d.%d\n", __clang_major__, __clang_minor__,
                __clang_patchlevel__);
#elif defined(__GNUC__)
    std::printf("compiler: g++ %d.%d.%d\n", __GNUC__, __GNUC_MINOR__,
                __GNUC_PATCHLEVEL__);
#else
    std::printf("compiler: not GCC or Clang\n");
#endif
    std::printf("C++ standard: %ld\n", static_cast<long>(__cplusplus));
    std::printf("bytes: int %zu, long %zu, long long %zu, pointer %zu\n",
                sizeof(int), sizeof(long), sizeof(long long), sizeof(void*));
    std::printf("INT_MAX   = %d\n", INT_MAX);
    std::printf("LLONG_MAX = %lld\n", LLONG_MAX);
    return 0;
}
```

The lines that start with `#if` are read by the *preprocessor* before the compiler proper: each compiler defines a few names that identify it, such as `__clang__` for Clang and `__GNUC__` for GCC, and only the branch that matches is compiled. The name `__cplusplus` holds the language standard as a number, `201703` for C++17. Compile and run it in the terminal of VS Code, from the `Chapter01` folder:

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o cpp/check_env cpp/check_env.cpp
$ ./cpp/check_env
```

```
compiler: g++ 13.3.0
C++ standard: 201703
bytes: int 4, long 8, long long 8, pointer 8
INT_MAX   = 2147483647
LLONG_MAX = 9223372036854775807
```

Read the third line carefully, because it hides a classic trap. On Linux and macOS, `long` has 8 bytes; on Windows it has only 4, the same as `int`, even with a 64-bit compiler. A program that stores a large number in a `long` works on Linux and overflows on Windows. The type `long long` has 8 bytes everywhere, and it is the type this book uses whenever a value can exceed about two billion, the limit `INT_MAX` printed on the fourth line. Equation (1.2) states the two limits exactly.

$$\text{INT\_MAX} = 2^{31} - 1 = 2\,147\,483\,647, \qquad \text{LLONG\_MAX} = 2^{63} - 1 \approx 9.2 \times 10^{18} \tag{1.2}$$

The Python version, **Listing 1.3**, reports the interpreter instead.

Listing 1.3 `check_env.py`: reports the Python version and shows that Python's integers have no fixed size.

```python
"""check_env.py: shows which Python runs this program and how big its ints get."""
import platform
import sys


def main():
    print("python:", platform.python_version(), platform.python_implementation())
    print("system:", platform.system(), platform.machine())
    print("sys.maxsize =", sys.maxsize)
    print("2**100 =", 2**100)
    print("inside a virtual environment:", sys.prefix != sys.base_prefix)


if __name__ == "__main__":
    main()
```

```
$ python3 python/check_env.py
```

```
python: 3.14.7 CPython
system: Linux x86_64
sys.maxsize = 9223372036854775807
2**100 = 1267650600228229401496703205376
inside a virtual environment: False
```

The fourth line shows why Python is so pleasant for problems with big numbers: 2^{100} has 31 digits, and Python computes it exactly without being asked. `sys.maxsize` is not the largest integer, since there is none, but the largest size of a list, which on a 64-bit machine happens to equal `LLONG_MAX`. The last line says whether a *virtual environment*, a private folder of installed packages, is active; the programs of Chapters 1 to 4 use only the standard library and need none.

1.11 Your First SPOJ Problem: TEST

The problem (SPOJ TEST, "Life, the Universe, and Everything", `https://www.spoj.com/problems/TEST/`) [3]. The input is a sequence of integers, one per line; each has at most two digits. Print the numbers one per line, in the same order, until you read the number 42; print neither 42 nor anything after it. The problem does not say how many numbers there are, and nothing guarantees that the input ends right after the 42: numbers may follow it, and the program must ignore them.

The sample input on the problem page is the five numbers 1, 2, 88, 42, 99, and the sample output is 1, 2, 88.

The idea. Read a number; if it is 42, stop; otherwise print it and repeat. There is nothing to prove. What there is to get right is the order of the three actions, *read, test, print*, and the behavior at the end of the input, which is where the beginners' bugs live.

Listing 1.4 `test_life.cpp`: the solution of TEST in C++.

```cpp
// SPOJ TEST (Life, the Universe, and Everything): echo numbers until 42.
#include <iostream>

int main() {
    int x;
    // The read comes first: the loop stops at 42 or when the input ends.
    while (std::cin >> x && x != 42) {
        std::cout << x << '\n';
    }
    return 0;
}
```

The whole program is the condition of the `while` loop. The expression `std::cin >> x` reads one integer into `x` and then *evaluates to the stream itself*; a stream used as a condition is true when the last read succeeded and false when it failed, for instance because the input has ended. The operator `&&` evaluates its right side only when the left side is true, so `x != 42` is tested only on a number that was really read. The loop therefore stops in exactly two situations, which are the two that the problem allows: the number 42 was read, or the input ended. Writing `'\n'` rather than `std::endl` ends the line without forcing the output out immediately; Chapter 4 measures why that matters.

Listing 1.5 `test_life.py`: the solution of TEST in Python.

```python
"""SPOJ TEST (Life, the Universe, and Everything): echo numbers until 42."""
import sys


def main():
    for line in sys.stdin:
        line = line.strip()
        if not line:  # skip empty lines, if any
            continue
        x = int(line)
        if x == 42:
            break
        print(x)


if __name__ == "__main__":
    main()
```

The Python version reads the input line by line with `for line in sys.stdin`, which also stops by itself at the end of the input. It removes the line break with `strip()` and skips empty lines, which the problem does not promise but which cost nothing to handle. The last two lines are a Python convention: `main()` runs only when the file is started as a program, not when another program imports it, which will let the test tools of later chapters reuse functions from our solutions.

Both programs give the same result on the sample. Compile the C++ version with **Ctrl+Shift+B** or with the command below, and feed both the sample:

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o cpp/test_life cpp/test_life.cpp
$ printf '1\n2\n88\n42\n99\n' | ./cpp/test_life
$ printf '1\n2\n88\n42\n99\n' | python3 python/test_life.py
```

```
1
2
88
```

(Each program printed these three lines; the output is shown once.) The command `printf` writes its argument, with `\n` turned into line breaks, and the pipe `|` connects it to the program's standard input. In PowerShell, `"1","2","88","42","99" | .\cpp\test_life.exe` does the same.

1.12 Building a Judge of Your Own

Rina's midnight message came from a gap in her workflow: she had no quick way to try her program on more than one input and compare the results mechanically. Comparing by eye fails exactly when you are tired, and it fails silently. The remedy is to do on your own computer what the judge does, which **Figure 1.1** already describes completely: compile once, run on each test with a time limit, compare with the expected output, stop at the first failure. It takes fewer than a hundred lines of Python.

1.12.1 Tests as pairs of files

A test is two files with the same name: `01.in`, the input, and `01.out`, the output a correct program prints for it. A folder of such pairs is a test set. Where do the expected outputs come from? For a problem as simple as TEST we can write them with a few lines of obviously correct code. **Listing 1.6** writes four tests: the sample, a list that starts with 42, a list with a zero, and a long list of five thousand numbers with more numbers after the 42.

Listing 1.6 `gen_test_life.py`: writes four tests for TEST, with the expected outputs computed by the simplest possible code.

```
"""gen_test_life.py: writes the tests of TEST into tests/TEST (.in and .out)."""
import random
from pathlib import Path

HERE = Path(__file__).resolve().parent / "TEST"


def expected(numbers):
    """The slow, obvious answer: everything before the first 42."""
    return numbers[:numbers.index(42)]


def write(name, numbers):
    HERE.mkdir(exist_ok=True)
    (HERE / f"{name}.in").write_text("".join(f"{x}\n" for x in numbers))
    (HERE / f"{name}.out").write_text("".join(f"{x}\n" for x in expected(numbers)))


def main():
    rng = random.Random(2026)
    write("01", [1, 2, 88, 42, 99])  # the sample of the problem
    write("02", [42, 1, 2])  # 42 first: print nothing
    write("03", [7, 0, 42])  # a zero before the end
    other = [x for x in range(100) if x != 42]
    long_run = [rng.choice(other) for _ in range(5000)]
    write("04", long_run + [42] + [rng.randrange(100) for _ in range(20)])
    print("wrote", len(list(HERE.glob("*.in"))), "tests to", HERE.name)


if __name__ == "__main__":
    main()
```

The function `expected` is the specification of the problem written as code: everything before the first 42. The random numbers come from `random.Random(2026)`, a generator with a fixed *seed*, so that every run produces the same tests; random but repeatable inputs are what make a failure reproducible. Test `02` exists because an empty output is a legal answer that a careless program might get wrong, and test `04` because a program that works on five numbers may fail on five thousand.

```
$ cd tests
$ python3 gen_test_life.py
```

```
wrote 4 tests to TEST
```

1.12.2 The judge

Listing 1.7 is the judge itself.

Listing 1.7 `minijudge.py`: compiles a C++ program (or runs a Python one) on every test of a folder, with a time limit, and prints a verdict the way SPOJ does.

```python
"""minijudge.py: a small local judge that works the way SPOJ's does.

usage: python3 minijudge.py TESTDIR SECONDS PROGRAM
PROGRAM is a .cpp file (compiled first, like a judge does) or a .py file.
Every test is a pair NAME.in / NAME.out in TESTDIR; tests run in name order
and judging stops at the first test that fails.
"""
import os
import subprocess
import sys
import tempfile
import time
from pathlib import Path

CXX = os.environ.get("CXX", "g++")
FLAGS = ["-std=c++17", "-O2", "-Wall", "-Wextra"]


def compile_cpp(source, workdir):
    """Compile source; return (command, "") or (None, compiler messages)."""
    exe = Path(workdir) / ("prog.exe" if os.name == "nt" else "prog")
    result = subprocess.run([CXX, *FLAGS, "-o", str(exe), source],
                            capture_output=True, text=True)
    if result.returncode != 0:
        return None, result.stderr
    return [str(exe)], ""


def same_output(got, expected):
    """Equal line by line, ignoring spaces at line ends and blank lines at end."""
    a = [line.rstrip() for line in got.rstrip().splitlines()]
    b = [line.rstrip() for line in expected.rstrip().splitlines()]
    return a == b


def first_difference(got, expected):
    a, b = got.splitlines(), expected.splitlines()
    for i in range(max(len(a), len(b))):
        x = a[i].rstrip() if i < len(a) else "<nothing>"
        y = b[i].rstrip() if i < len(b) else "<nothing>"
        if x != y:
            return f"line {i + 1}: got {x!r}, expected {y!r}"
    return "only blank lines differ"
```

Listing 1.7 (Continued) *minijudge.py: compiles a C++ program (or runs a Python one) on every test of a folder, with a time limit, and prints a verdict the way SPOJ does.*

```
def run_one(command, infile, seconds):
    """Run once with infile as standard input; return (status, time, output)."""
    with open(infile, "rb") as f:
        start = time.perf_counter()
        try:
            r = subprocess.run(command, stdin=f, capture_output=True,
                               timeout=seconds)
        except subprocess.TimeoutExpired:
            return "TLE", seconds, ""
        used = time.perf_counter() - start
    if r.returncode != 0:
        lines = r.stderr.decode(errors="replace").strip().splitlines()
        return "RE", used, lines[-1] if lines else f"exit code {r.returncode}"
    return "OK", used, r.stdout.decode(errors="replace")


def judge(testdir, seconds, program):
    tests = sorted(Path(testdir).glob("*.in"))
    with tempfile.TemporaryDirectory() as work:
        if program.endswith(".cpp"):
            command, errors = compile_cpp(program, work)
            if command is None:
                print("compilation error")
                lines = errors.splitlines()
                print([x for x in lines if "error" in x][0] if lines else "")
                return "CE"
        else:
            command = [sys.executable, program]
        for test in tests:
            status, used, out = run_one(command, test, seconds)
            if status == "OK":
                expected = test.with_suffix(".out").read_text()
                status = "AC" if same_output(out, expected) else "WA"
            print(f"  test {test.stem}: {status:3} {used:5.2f} s")
            if status == "WA":
                print("    ", first_difference(out, expected))
            if status == "RE":
                print("    ", out)
            if status != "AC":
                return status
    return "AC"
```

Listing 1.7 (Continued) *minijudge.py: compiles a C++ program (or runs a Python one) on every test of a folder, with a time limit, and prints a verdict the way SPOJ does.*

```
def main():
    if len(sys.argv) != 4:
        sys.exit(__doc__)
    testdir, seconds, program = sys.argv[1], float(sys.argv[2]), sys.argv[3]
    verdict = judge(testdir, seconds, program)
    names = {"AC": "accepted", "WA": "wrong answer", "TLE": "time limit exceeded",
             "RE": "runtime error", "CE": "compilation error"}
    print(f"verdict: {verdict} ({names[verdict]})")


if __name__ == "__main__":
    main()
```

Walk through it in the order of **Figure 1.1**. The function `compile_cpp` runs the compiler with the book's flags, exactly as the judge compiles once before any test; if the compiler fails, `judge` prints the first line of the compiler's message that contains the word *error* and returns CE. For a Python

file there is nothing to compile, and the command becomes the current Python interpreter (`sys.executable`) followed by the file name.

The function `run_one` starts the program once with the test's `.in` file as its standard input and catches its standard output. The argument `timeout` of `subprocess.run` implements the time limit: when it expires, Python kills the program and raises `TimeoutExpired`, which becomes TLE. A program that ends with a non-zero exit code has crashed, which becomes RE, and the last line of whatever it wrote to its error stream is kept, because for Python that line names the exception.

The function `same_output` implements the comparison of step 3. It removes spaces at the end of every line and blank lines at the end of the output before comparing, which is roughly how SPOJ's default comparison treats whitespace; a line that differs in any other character is a difference. When the verdict is WA, `first_difference` shows the first line that differs, which is the single most useful piece of information when you hunt a wrong answer, and one the real judge never gives you.

Finally, `judge` runs the tests in name order and returns at the first failure, as SPOJ does. The time printed for each test is the wall-clock time measured by the judge, including the time to start the program; for Python that start-up alone takes a few hundredths of a second.

1.12.3 Judging correct and wrong programs

Now we can play judge. First the two solutions of Section 1.11, from the `Chapter01` folder:

```
$ python3 python/minijudge.py tests/TEST 1 cpp/test_life.cpp
$ python3 python/minijudge.py tests/TEST 1 python/test_life.py
```

```
  test 01: AC   0.01 s
  test 02: AC   0.00 s
  test 03: AC   0.00 s
  test 04: AC   0.01 s
verdict: AC (accepted)
  test 01: AC   0.02 s
  test 02: AC   0.03 s
  test 03: AC   0.02 s
  test 04: AC   0.04 s
verdict: AC (accepted)
```

The times on your computer will differ a little from run to run; the verdicts never do. Now the program Rina submitted, **Listing 1.8**.

Listing 1.8 `test_wa.cpp`: a wrong solution that tests for 42 only after printing it.

```cpp
// A wrong version of TEST: it checks for 42 only after printing the number.
#include <iostream>

int main() {
    int x;
    while (std::cin >> x) {
        std::cout << x << '\n';
        if (x == 42) break;
    }
    return 0;
}
```

```
$ python3 python/minijudge.py tests/TEST 1 cpp/test_wa.cpp
```

```
  test 01: WA   0.00 s
    line 4: got '42', expected '<nothing>'
verdict: WA (wrong answer)
```

The judge found the bug on the very first test, which is the sample, and named the line: the program printed a fourth line, `42`, where the correct output has ended. Rina missed it because the extra line is right where the eye stops reading. The order *read, print, test* instead of *read, test, print* is one of the most common bugs in loops of this shape, and a mechanical comparison finds it instantly.

The next wrong program, **Listing 1.9**, is subtler: it stores the numbers in a list of fixed size 100 before printing them, a habit carried over from languages with fixed-size arrays.

Listing 1.9 `test_re.py`: a wrong solution that keeps the numbers in a list of fixed size and crashes on long inputs.

```
"""A wrong version of TEST: it keeps the numbers in a list of fixed size 100."""
import sys


def main():
    kept = [0] * 100
    count = 0
    for line in sys.stdin:
        x = int(line)
        if x == 42:
            break
        kept[count] = x  # fails with IndexError at the 101st number
        count += 1
    for i in range(count):
        print(kept[i])


if __name__ == "__main__":
    main()
```

```
  $ python3 python/minijudge.py tests/TEST 1 python/test_re.py
```

```
  test 01: AC   0.02 s
  test 02: AC   0.04 s
  test 03: AC   0.03 s
  test 04: RE   0.09 s
    IndexError: list assignment index out of range
verdict: RE (runtime error)
```

It passes the three short tests and crashes on the long one, with the exception that Python names in its last line. On SPOJ this would read *runtime error (NZEC)*, because an uncaught Python exception ends the program with a non-zero exit code. Notice that the statement never said how many numbers there are; the program assumed a number, and a test with more numbers broke the assumption. This is why a good test set always contains a large test.

The last one, **Listing 1.10**, does not even compile.

Listing 1.10 `test_ce.cpp`: a solution with a missing semicolon.

```
// A version of TEST that does not compile: one semicolon is missing.
#include <iostream>

int main() {
    int x;
    while (std::cin >> x && x != 42) {
        std::cout << x << '\n'
    }
    return 0;
}
```

```
  $ python3 python/minijudge.py tests/TEST 1 cpp/test_ce.cpp
```

```
compilation error
cpp/test_ce.cpp:7:31: error: expected ‘;’ before ‘}’ token
verdict: CE (compilation error)
```

The compiler reports the file, the line (7) and the column (31) where it noticed the problem, which is the end of line 7, where the semicolon is missing. You would of course see this on your own computer before submitting; on the judge, a CE usually has a different cause, such as a feature that is newer than the judge’s compiler, or choosing the wrong language in the submission form.

The one verdict we have not produced is TLE. Chapter 3 produces it honestly, with a correct program whose algorithm is too slow, and the same `minijudge.py`.

1.13 Using It for Real

Everything so far happened on your computer. Here is how to take it to SPOJ, and how to build a habit around it.

Submitting. Create an account at `https://www.spoj.com` (the **sign up** link at the top). Open a problem, for example `https://www.spoj.com/problems/TEST/`, and click **Submit**. Choose the language from the list, and choose it carefully: for C++ choose the entry that names the newest GCC offered, which at the time of writing is “C++14 (gcc 8.3)” [4]; for Python choose “Python 3”, never Python 2, whose `print` and division work differently. Paste the source or upload the file, and press **Send**. The status page shows the verdict within seconds, together with the running time and memory, and your submission stays in the list of your submissions, where you can open it again.

Before every submission, run your own judge. For a new problem, make a `tests` folder with the sample as `01.in`/`01.out`, add the smallest case you can think of, and one large case. Chapter 2 shows how to make large cases with a generator, and how to get their expected outputs from a brute force when you cannot write them by hand. Two minutes spent here save many red verdicts and, more importantly, teach you to predict where a program breaks.

Keep a notebook. For every problem you solve, write down its code, the idea in one sentence, and every verdict you received with its cause: “ADDREV: WA, printed leading zeros.” After fifty problems the notebook becomes a personal list of your favorite mistakes, which is the most valuable debugging tool you will own. The Common Mistakes sections of this book are such a list, collected from many people.

For a school or campus club. `minijudge.py` works for any problem whose output is compared line by line. A teacher can give a club a folder of tests for a problem, and every member can judge their program locally, without an internet connection and without waiting for a queue, before submitting to SPOJ. Changing the flags in `FLAGS` to `-std=c++14` makes it reject exactly the programs that the judge’s compiler would reject.

A rhythm that works. Competitive programming rewards regular practice more than long weekends. Solving one or two problems a day, reading the statement properly, predicting the complexity before coding, testing locally and then submitting, builds the skills of this book far faster than reading about them. When a problem has resisted you for an hour, read a hint or an explanation of the idea, then close it and write the program yourself: a solution you typed after understanding it stays with you; a solution you copied does not.

1.14 Common Mistakes, and How to Avoid Them

- **Trusting the sample.** Passing the sample shows that the format is right and little else. Always add at least the smallest input and the largest input before submitting.
- **Checking after acting.** In a loop that must stop at a sentinel value such as 42, the order is *read, test, act*. `test_wa.cpp` acts first.
- **Assuming a fixed input size.** When a statement does not bound the number of lines, do not invent a bound. Read until the input ends, and store only what you need.
- **Printing prompts.** A judge compares every character; a friendly `Enter a number:` is a wrong answer. Print exactly what the output format asks for, nothing more.
- **Using `long` for big numbers.** It has 4 bytes on Windows. Use `long long`, as Equation (1.2) and `check_env.cpp` show.
- **Including `<bits/stdc++.h>` on a Mac.** Apple's Clang does not have it. Include the standard headers you need, or install GCC with Homebrew.
- **Choosing the wrong language on SPOJ.** C++ code submitted as C, or Python 3 code submitted as Python 2, gives CE or RE for no visible reason. Check the language before pressing Send.
- **Typing `<` in PowerShell.** It is refused. Use Command Prompt, `Get-Content file | program`, or the `minijudge.py` of this chapter, which feeds the files itself.
- **Reading the verdict as a grade.** TLE says nothing about correctness, and WA nothing about speed. Use **Table 1.1** to decide where to look.

1.15 Chapter Summary

- Competitive programming is solving exactly specified problems with programs that a judge checks for correct output within a time limit and a memory limit. SPOJ is a large permanent archive of such problems, each with a short code such as TEST; its classical section gives a binary verdict.
- A judge compiles the source once, runs it on each hidden test with its standard input connected to the test file, compares the output with the expected one, and stops at the first failure (**Figure 1.1**). The verdicts AC, WA, TLE, RE and CE each point to a different kind of bug (**Table 1.1**).
- The time a program needs is roughly its number of steps divided by the steps per second of the machine (Equation (1.1)), about 10^8 to 10^9 for C++ and far fewer for Python.
- The workbench of this book is g++ (MSYS2 on Windows, the Command Line Tools or Homebrew on macOS, `build-essential` on Linux), Python 3.14 and VS Code with the C/C++ and Python extensions (**Table 1.2**), with a build task that compiles with `-std=c++17 -O2 -Wall -Wextra` (**Listing 1.1**). Every program also compiles as C++14, so that it suits SPOJ's older compiler.
- `long` has 4 bytes on Windows; `long long` holds up to about 9.2×10^{18} everywhere (Equation (1.2)). Python's integers have no limit.
- TEST is solved by a loop that reads, tests for 42, and only then prints, in C++ (**Listing 1.4**) and Python (**Listing 1.5**).
- A local judge (**Listing 1.7**) with a folder of tests (**Listing 1.6**) gives the same verdicts as SPOJ in a fraction of a second and shows the first differing line. It caught a wrong order of actions (**Listing 1.8**), a fixed-size list (**Listing 1.9**) and a syntax error (**Listing 1.10**).

1.16 Exercises

1. For each situation, name the verdict SPOJ gives and the first thing you would check: (a) the program prints `Case 1: 5` where the output format asks for `5`; (b) a C++ program reads a million numbers into `int a[1000]`; (c) a correct program loops over all numbers from 1 to 10^{12}; (d) a program written with a C++20 feature is submitted as "C++14 (gcc 8.3)".
2. Use Equation (1.1) to estimate how long a C++ program needs for 3×10^9 simple steps, and whether it fits a one-second limit. How long would a Python program need for the same number of steps, at 10^7 steps per second?
3. Write a test `05` for TEST whose input contains the number 42 twice, and explain what the correct output is. Which of the programs of this chapter pass it?
4. Change `minijudge.py` so that, instead of stopping at the first failure, it runs every test and prints a summary such as `3 of 4 tests passed`. Why do real judges stop at the first failure?
5. Run `check_env.cpp` on your own computer and compare its output with the one in Section 1.10. On which systems does the third line differ, and why does that matter for a program that stores the number 3×10^9?
6. A friend of Rina's says: "Your program passed the sample, so the judge must be wrong." Write, in three or four sentences, the answer that this chapter suggests.

1.17 Solutions to the Exercises

1. (a) WA: the output format was not followed; print exactly `5`. (b) RE, most likely `SIGSEGV`, because the program writes past the end of the array; size the array from the constraints or use a `std::vector`. (c) TLE: 10^{12} steps need well over a thousand seconds by Equation (1.1); the algorithm must change, not the code around it. (d) CE: the judge's compiler does not know the feature; compile locally with `-std=c++14` before submitting.

2. At 10^8 to 10^9 steps per second, 3×10^9 steps need between 3 and 30 seconds, so the program does not fit a one-second limit even on a fast machine. At 10^7 steps per second Python needs $3 \times 10^9 / 10^7 = 300$ seconds, five minutes.

3. Input `5`, `42`, `7`, `42`, `9`, one per line; the correct output is the single line `5`, because the program must stop at the *first* 42. The expected file `05.out` therefore contains `5`. Both correct programs, `test_life.cpp` and `test_life.py`, pass it, because they stop at the first 42 and never look further. `test_re.py` passes it too, since the input is short. `test_wa.cpp` fails, printing `5` and `42`. To add the test, append `write("05", [5, 42, 7, 42, 9])` to `main` in `gen_test_life.py`; the function `expected` already cuts at the first 42.

4. In `judge`, remove the line `if status != "AC": return status`, count the tests with verdict AC in a variable `passed`, and after the loop print `f"{passed} of {len(tests)} tests passed"` and return `"AC"` only if `passed == len(tests)`, otherwise the first failing verdict, which you remember in a variable. Real judges stop at the first failure for two reasons: it saves machine time when thousands of submissions wait in the queue, and it avoids telling the contestant which tests pass, which would let them probe the hidden tests.

5. The line reads `long 8` on Linux and macOS and `long 4` on Windows, with both MSYS2's g++ and Microsoft's compiler. The number 3×10^9 is larger than `INT_MAX`, $2^{31} - 1 \approx 2.1 \times 10^9$ (Equation (1.2)), so a 4-byte `long` cannot hold it: the program is correct on Linux and wrong on Windows. `long long` holds it on every system.

6. The sample only shows the format, and the judge runs the program on many hidden tests that were designed to find mistakes: the smallest and largest inputs, special values, and long inputs. The judge does not read code, it compares output, so a program that prints one extra line fails even if it is "almost right". The right response to WA is to write more tests, compare mechanically with a tool like `minijudge.py`, and look for the input on which the program misbehaves.

References

[1] ICPC Foundation, "The ICPC International Collegiate Programming Contest." [Online]. Available: https://icpc.global (accessed Sep. 2026).

[2] Sphere Research Labs, "SPOJ: Sphere Online Judge." [Online]. Available: https://www.spoj.com (accessed Sep. 2026).

[3] SPOJ, "TEST — Life, the Universe, and Everything." [Online]. Available: https://www.spoj.com/problems/TEST/ (accessed Sep. 2026).

[4] hkktr, "polski-spoj: solutions for POLSKI SPOJ problems," README, stating the language used as "C++14 (gcc 8.3)". [Online]. Available: https://github.com/hkktr/polski-spoj (accessed Sep. 2026).

[5] GNU Project, "GCC, the GNU Compiler Collection," release list with GCC 16.2 (Aug. 7, 2026) and GCC 15.3 (Jun. 12, 2026). [Online]. Available: https://gcc.gnu.org/ (accessed Sep. 2026).

[6] L. Rendek, "How to Install GCC and Compile C/C++ Programs on Ubuntu 26.04," LinuxConfig, Apr. 1, 2026. [Online]. Available: https://linuxconfig.org/how-to-install-gcc-and-compile-c-c-programs-on-ubuntu-26-04

[7] MSYS2 Developers, "Package: mingw-w64-ucrt-x86_64-gcc," version 16.2.0-4, built Sep. 20, 2026. [Online]. Available: https://packages.msys2.org/packages/mingw-w64-ucrt-x86_64-gcc

[8] Apple Inc., "Installing the command-line tools," *Xcode Documentation*. [Online]. Available: https://developer.apple.com/documentation/xcode/installing-the-command-line-tools (accessed Sep. 2026).

[9] Python Software Foundation, "Python 3.14.7," Aug. 5, 2026. [Online]. Available: https://www.python.org/downloads/latest/python3.14/

[10] Microsoft, "Visual Studio Code Updates, version 1.139," Sep. 23, 2026. [Online]. Available: https://code.visualstudio.com/updates

[11] Microsoft, "C/C++ for Visual Studio Code," *Visual Studio Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs/languages/cpp (accessed Sep. 2026).

[12] Microsoft, "Python in Visual Studio Code," *Visual Studio Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs/languages/python (accessed Sep. 2026).

[13] Microsoft, "Using GCC with MinGW," *Visual Studio Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs/cpp/config-mingw (accessed Sep. 2026).

[14] MSYS2 Developers, "MSYS2: Software distribution and building platform for Windows." [Online]. Available: https://www.msys2.org (accessed Sep. 2026).

[15] Python Software Foundation, "Using Python on Windows," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/using/windows.html (accessed Sep. 2026).

[16] Homebrew, "gcc: GNU compiler collection," formula version 16.2.0. [Online]. Available: https://formulae.brew.sh/formula/gcc (accessed Sep. 2026).

Chapter 2

Reading the Problem Like a Judge Does

Most rejected submissions are not failures of algorithmic skill but failures of reading. A word such as “at most”, a sentence such as “omit leading zeros”, a line in the constraints that says the input can hold a hundred cases, or a priority list that is not the one you learned at school: each of them decides whether a correct idea turns into an accepted program. This chapter teaches a disciplined way to read a statement before writing any code. By the end of it you will be able to find the six zones of a problem statement and say what to take from each; answer a short reading checklist that catches most wrong answers before they happen; recognize the three shapes in which judge input arrives and read each of them correctly in C++ and Python; restate a problem in your own words, including its traps; and solve three real SPOJ problems, ADDREV, FCTRL2 and ONP, each checked against a brute force by a stress test that you will reuse for the rest of the book.

2.1 The Line Nobody Read

Every Indonesian who has renewed a driving licence or filled in a tax form knows the feeling: the form is returned at the counter because a box was filled in the wrong format, a date written as day-month where month-day was expected, a name in lower-case letters where capitals were required. The officer is not being cruel. The instructions were printed on the form; they were simply not read, because everybody knows how to write a date.

Rina’s campus team met the same officer in their first online contest. Their program for a prime-number problem was correct and fast. It was rejected with *wrong answer* three times. The statement, near its end, in a sentence that nobody on the team had read to the last word, asked for an empty line between the outputs of consecutive test cases. The team printed the primes of every case one after another without it. After the contest the coach made them read the statement aloud, one sentence at a time, and write down every rule they heard. They found the empty line in the second minute.

The lesson is not “be more careful”, which is advice nobody can follow. The lesson is that reading a statement is a skill with a procedure, like reading a map or a contract, and that the procedure can be learned. A judge reads your output against the statement with perfect attention to every character. The only defense is to read the statement with the same attention, once, systematically, before the first line of code.

Who benefits? Everybody who writes programs against a specification, which is every programmer. A developer who implements an interface described by a bank, a tax office or a payment gateway faces exactly this kind of text: a format for the input, a format for the output, limits on sizes, rules for special cases, and examples that illustrate the format without covering the rules. Reading such a document the way this chapter teaches is a large part of doing the job well.

2.2 The Anatomy of a Problem Statement

Nearly every statement on SPOJ, and on every other judge, is built from the same six zones. They do not always carry headings, and they are not always in the same order, but they are always there, and **Figure 2.1** lists them together with what you should write down from each.

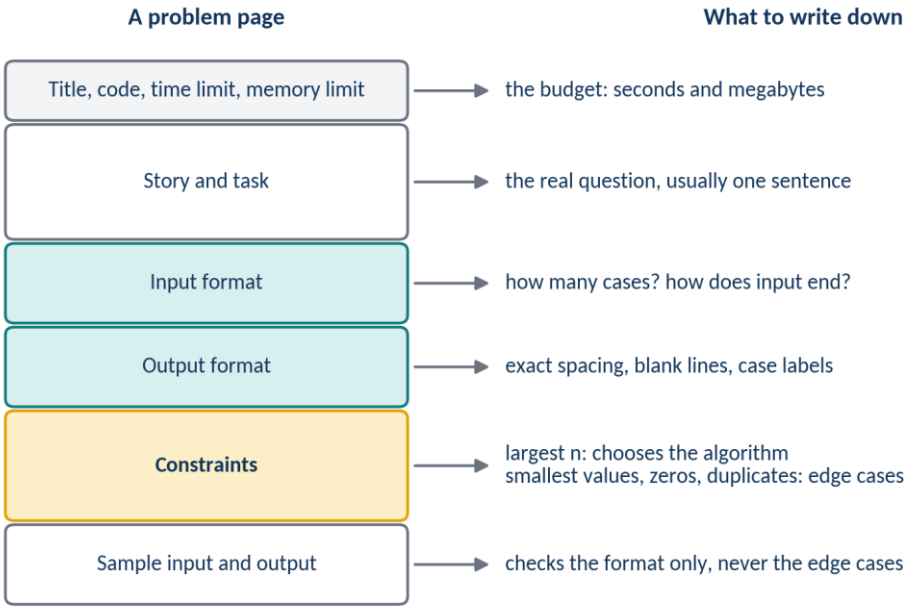


Figure 2.1 The six zones of a problem statement and what to write down from each. The constraints, marked in amber, decide both the algorithm and the edge cases.

Header: title, code, limits. The top of every SPOJ problem page shows the problem code, the time limit per test file and the memory limit, and often the list of languages that are allowed. These are part of the specification, as much as the input format. Write down the time limit, because Chapter 3 turns it into a budget of steps.

Story and task. Many statements wrap the task in a story: actors, cows, a king with a treasure. The story makes the problem memorable, and it sometimes disguises a standard problem as something unfamiliar. The *task* is usually one sentence, often the last sentence of the story, that says what must be computed. Find that sentence and underline it; everything else in the story is decoration, unless it defines a term that the task uses.

Input format. This zone is a contract. It says how many numbers or lines come, in which order, and how the input ends. “The first line contains the number of test cases” and “the input ends at end of file” require different loops, and Section 2.4 shows both.

Output format. Another contract, and the one whose violations are most frustrating, because they produce WA on a correct algorithm. Look for exact spelling, case labels such as `Case 1:`, separators between numbers, empty lines between cases, the number of digits after a decimal point, and what to print when there is no answer.

Constraints. The real specification. The largest values say which algorithms are fast enough, as Chapter 3 shows in detail. The smallest values, and the words around them, tell you which edge cases exist: can n be 0 or 1? can numbers be negative, equal, or zero? is the input already sorted? do values fit in 32 bits? A statement that says $1 \le n$ has just told you that you never need to handle an empty list; one that says $0 \le n$ has told you that you must.

Samples. An example input and the output for it. The sample shows the format, and working it by hand is the best check that you understood the task. It almost never covers the edge cases, and it is never the full test set.

> **KEY INSIGHT.** Read the constraints twice: once for size, once for shape. The first pass answers "how large can the input get?", which chooses the algorithm. The second pass answers "what can the values look like?", zero, negative, equal, huge, which chooses the edge cases to test. Most beginners do only the first pass.

To see the two passes at work, take the constraints of PRIME1, the problem Rina's team stumbled over, which Chapter 3 solves in full: at most 10 test cases, each a range from m to n with $1 \le m \le n \le 10^9$ and $n - m \le 10^5$. The size pass reads: numbers up to a billion, but each range holds at most a hundred thousand of them, and there are at most ten ranges, so at most a million numbers must be examined in total; an algorithm that does a little work per number of the range is fine, one that touches every number up to n is not. The shape pass reads: m can be 1, which is not a prime, so the program must not print it; m can equal n, so a range can hold a single number; and the numbers reach 10^9, which fits in an `int`, while a product such as $p \cdot p$ for p near 31 623 is already close to the limit of `int` and must be computed in `long long`. Six short observations, and every one of them corresponds to a bug that someone has submitted. Add the output rule that Rina's team missed, an empty line between test cases, and the whole specification fits on an index card.

2.3 A Reading Checklist

A checklist turns attention into a procedure. **Table 2.1** has eight questions; answer each of them, in writing if the problem is new to you, before you write code. Next to each question is the kind of verdict that skipping it usually produces.

Table 2.1 A reading checklist: eight questions to answer before writing code, and the verdict that skipping each one tends to produce.

#	Question	Where to look	Skipping it gives
1	What exactly is asked? Can I say it in one sentence?	Story and task	WA on everything
2	How many test cases, and how does the input end?	Input format	WA, RE or TLE waiting for input
3	What is the largest input, and what complexity does it allow?	Constraints, time limit	TLE
4	What is the smallest input, and can values be 0, negative, equal?	Constraints	WA on edge cases, RE
5	Do all values and intermediate results fit in `int`? in `long long`?	Constraints	WA from overflow
6	What exactly must be printed, with which separators and empty lines?	Output format	WA on a correct algorithm
7	Is the answer unique? If not, what must be printed?	Task, output format	WA from a different valid answer
8	Does my hand calculation of the sample give the sample output?	Samples	WA from a misunderstood task

The best way to answer question 1 is to write a *restatement*: two to five sentences, in your own words, that say what the input is, what the output is, and every special rule. A restatement is useful exactly because it forces you to decide what each sentence of the original means. If you cannot write it, you have not understood the problem yet, and no amount of coding will fix that.

From now on, every SPOJ problem in this book begins with such a restatement, in the style you should use in your own notebook. The standard training books of competitive programming recommend the same discipline of careful reading, modeling and testing [1], [2].

A few traps recur often enough on SPOJ to be named.

- **"At most" versus "exactly".** "At most 100 test cases" allows one; "at most 10 digits" allows one digit. Your loops must not assume the maximum.
- **Rules stated once.** A statement that says in its first paragraph that positions are numbered from 1 will not say it again in the constraints.
- **Hidden units.** A time in minutes, an angle in degrees, money in cents: the story usually says it, the input format does not.
- **Ties.** "Print the smallest", "print any", or silence: three different problems.
- **Ranges and their ends.** "From a to b" usually includes both ends, "between" is ambiguous in everyday language, and "the first k" may count from 0 or from 1. When the statement is precise, as with "inclusive", obey it; when it is not, the sample usually decides, so work it by hand.
- **Samples that hide the rule.** A sample with only positive, distinct, small values says nothing about zeros, duplicates or large values, even when the constraints permit all three.

2.4 The Three Shapes of Judge Input

Judge input comes in three shapes, and each needs its own loop.

Count first. The first line holds a number T, and T test cases follow. This is the most common shape on SPOJ, and ADDREV, FCTRL2 and ONP in this chapter all use it. The loop reads T and then runs exactly T times.

Sentinel. The input ends with a special value that is not itself a test case, such as the 42 of TEST in Chapter 1, or a line with `0 0`. The loop reads, compares with the sentinel, and stops.

Until end of file. No count and no sentinel: the test cases simply continue until the input ends. The loop must detect the end of the input itself. **Listing 2.1** shows the C++ idiom for a hypothetical problem that asks for the sum of each pair of numbers.

Listing 2.1 `sum_until_eof.cpp`: the "until end of file" input shape in C++.

```cpp
// Input shape "until end of file": add each pair of numbers, however many come.
#include <iostream>

int main() {
    long long a, b;
    int line = 0;
    while (std::cin >> a >> b) {  // false when the input is exhausted
        line++;
        std::cout << "pair " << line << ": " << a + b << '\n';
    }
    return 0;
}
```

As in Chapter 1, `std::cin >> a >> b` reads two numbers and evaluates to the stream, which is false once a read has failed; at the end of the input the read fails and the loop ends. There is no need to know in advance how many pairs there are, and no need for a separate check such as `std::cin.eof()`. That check is in fact a trap: `eof()` becomes true only *after* a read has tried to go past the end, so a loop `while (!std::cin.eof())` runs once too often and processes the last pair twice or a garbage pair once.

Listing 2.2 `sum_until_eof.py`: the same input shape in Python, by reading all tokens at once.

```python
"""Input shape "until end of file": add each pair of numbers, however many come."""
import sys


def main():
    numbers = [int(t) for t in sys.stdin.read().split()]
    for line, i in enumerate(range(0, len(numbers) - 1, 2), start=1):
        print(f"pair {line}: {numbers[i] + numbers[i + 1]}")


if __name__ == "__main__":
    main()
```

The Python version reads the whole input with `sys.stdin.read()`, splits it into *tokens* (maximal runs of characters without spaces or line breaks) with `split()`, and converts them to integers. This style has a large advantage that is easy to miss: it does not care how the numbers are distributed over lines. Both programs give the same answer whether the three pairs are written one pair per line or all on one line:

```
$ printf '1 2\n3 4\n100 200\n' | ./cpp/sum_until_eof
$ printf '1 2 3 4\n\n100\n200' | python3 python/sum_until_eof.py
```

```
pair 1: 3
pair 2: 7
pair 3: 300
```

(Each program printed these lines, for either input.) The second input has an empty line in the middle and no line break at the end, two things that break programs reading line by line and that real judge files do contain. Unless the line structure itself carries information, read tokens, not lines. The same token-based reading also protects you against the character `\r` that files written on Windows carry at the end of every line: `split()` treats it as whitespace, and so does `std::cin >>`.

2.5 ADDREV: Reading "Omit Leading Zeros"

The problem (SPOJ ADDREV, "Adding Reversed Numbers", `https://www.spoj.com/problems/ADDREV/`) [3]. The statement introduces *reversed numbers*: a number written with its digits in the opposite order, so that 1245 becomes 5421. When a number ends in zeros, its reversed form loses them, so 1200 reversed is 21, not 0021: a reversed number never has leading zeros. The input begins with the number of test cases; each case is a line with two positive integers, which are already *reversed* numbers. Take the two original numbers they came from, add them, and print the sum in reversed form, again without leading zeros. The statement also tells you that the original numbers are understood to be those without trailing zeros, since zeros lost in reversing cannot be recovered.

Restated in one sentence: *for each pair a, b, print* $\mathrm{rev}(\mathrm{rev}(a) + \mathrm{rev}(b))$*, where* rev *reverses the digits and drops leading zeros.*

Reading the traps. Work through the checklist. Question 2: count first. Question 4: the numbers are positive, so 0 never appears as input, but a sum can end in zeros, and then its reversal starts with zeros that must be dropped. Question 5: the input numbers are of modest size, but a sum can have one more digit than its operands, and we use 64-bit `long long` so that no case can overflow. Question 6: one number per line.

$$\text{rev}(x) = \sum_{i=0}^{k-1} d_i \cdot 10^{k-1-i} \quad \text{where } x = \sum_{i=0}^{k-1} d_i \cdot 10^{i} \tag{2.1}$$

Equation (2.1) says it precisely: if x has the digits $d_{k-1} \ldots d_1 d_0$ from left to right, so that d_0 is the ones digit, then $\text{rev}(x)$ has the digits $d_0 d_1 \ldots d_{k-1}$. When $d_0 = 0$, the leading digit of $\text{rev}(x)$ is 0, which simply means that the number has fewer digits; as an integer, the zero vanishes by itself.

A worked example. The third sample case is 305 and 794.

1. $\text{rev}(305) = 503$, and $\text{rev}(794) = 497$.
2. $503 + 497 = 1000$.
3. $\text{rev}(1000)$ is the digit string `0001`, which as a number is 1.

So the answer is `1`. This case is the whole problem in miniature, and the sample contains it, which is a kindness of the problem setter: the three sample cases are 24,1 with answer 34; 4358,754 with answer 1998; and 305,794 with answer 1.

The idea. Compute rev on integers, digit by digit, instead of on strings, as **Listing 2.3** does. Taking x mod 10 gives the last digit, and $\lfloor x/10 \rfloor$ removes it; appending a digit d to r is $r \leftarrow 10r + d$. Leading zeros of the result disappear automatically, because $10 \cdot 0 + 0 = 0$.

Listing 2.3 `addrev.cpp`: ADDREV with integer digit reversal.

```cpp
// SPOJ ADDREV (Adding Reversed Numbers): reverse(reverse(a) + reverse(b)).
#include <iostream>

// Digits of x in the opposite order; leading zeros of the result vanish.
long long reverseDigits(long long x) {
    long long r = 0;
    while (x > 0) {
        r = r * 10 + x % 10;
        x /= 10;
    }
    return r;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int cases;
    std::cin >> cases;
    for (int c = 0; c < cases; c++) {
        long long a, b;
        std::cin >> a >> b;
        std::cout << reverseDigits(reverseDigits(a) + reverseDigits(b)) << '\n';
    }
    return 0;
}
```

The loop in `reverseDigits` runs once per digit, so a case costs time proportional to the number of digits, at most about ten, and the whole program is linear in the size of the input. The two lines after the declaration of `main` make input and output faster; they are harmless here and Chapter 4 explains them.

Listing 2.4 `addrev.py`: ADDREV in Python, reversing through strings.

```
"""SPOJ ADDREV (Adding Reversed Numbers): reverse(reverse(a) + reverse(b))."""
import sys


def reverse_digits(x):
    """Digits of x in the opposite order; leading zeros of the result vanish."""
    return int(str(x)[::-1])


def main():
    data = sys.stdin.read().split()
    cases = int(data[0])
    out = []
    for c in range(cases):
        a = int(data[1 + 2 * c])
        b = int(data[2 + 2 * c])
        out.append(str(reverse_digits(reverse_digits(a) + reverse_digits(b))))
    print("\n".join(out))


if __name__ == "__main__":
    main()
```

The Python version, **Listing 2.4**, reverses through a string: `str(x)[::-1]` is the digits in reverse order, and `int()` of a string with leading zeros simply ignores them, so `int("0001")` is 1. It collects all answers in a list and prints them joined with line breaks, one large write instead of many small ones; Chapter 4 measures why that helps.

```
$ printf '3\n24 1\n4358 754\n305 794\n' | ./cpp/addrev
$ printf '3\n24 1\n4358 754\n305 794\n' | python3 python/addrev.py
```

```
34
1998
1
```

(Both programs printed these lines.)

2.5.1 A wrong version, and a tool that finds it

Listing 2.5 is the program that many people write first. It reverses the inputs correctly, but reverses the *sum* as a string and prints the string as it is.

Listing 2.5 `addrev_bug.cpp`: a wrong ADDREV that keeps the leading zeros of the reversed sum.

```cpp
// A wrong ADDREV: it reverses the sum as text and forgets the leading zeros.
#include <algorithm>
#include <iostream>
#include <string>

long long reverseDigits(long long x) {
    long long r = 0;
    while (x > 0) {
        r = r * 10 + x % 10;
        x /= 10;
    }
    return r;
}

int main() {
    int cases;
    std::cin >> cases;
    for (int c = 0; c < cases; c++) {
        long long a, b;
        std::cin >> a >> b;
        std::string s = std::to_string(reverseDigits(a) + reverseDigits(b));
        std::reverse(s.begin(), s.end());  // 1000 becomes "0001"
        std::cout << s << '\n';
    }
    return 0;
}
```

On the sample it prints `34`, `1998` and `0001`: the third line gives it away, if you look. But suppose the sample had not contained a sum ending in zero. Would you have found the bug? This is the moment to introduce the tool that finds such bugs systematically, the *stress test*. It needs three programs:

- a **generator** that prints one random input, determined by a seed given on the command line, so that any failure can be reproduced;
- a **brute force** that solves the problem in the simplest possible way, with no cleverness at all, so that its correctness is obvious;
- the **program under test**.

The stress test runs the generator with seeds 1, 2, 3, and so on, feeds each input to both programs, and stops at the first input on which their outputs differ. **Listing 2.6** is the stress test we use for the rest of the book.

Listing 2.6 `stress.py`: runs a generator, a brute force and a program on many seeds, and prints the first input on which the answers differ.

```
"""stress.py: compare a program with a brute force on many generated inputs.

usage: python3 stress.py GEN.py BRUTE.py PROGRAM [ROUNDS]
GEN.py prints one input for the seed given as its first argument.
BRUTE.py and PROGRAM read that input; PROGRAM may be a .cpp or a .py file.
The first input on which the two answers differ is printed, and the run stops.
"""
import os
import subprocess
import sys
import tempfile
from pathlib import Path

CXX = os.environ.get("CXX", "g++")
FLAGS = ["-std=c++17", "-O2", "-Wall", "-Wextra"]


def command_for(program, workdir):
    if program.endswith(".cpp"):
        exe = str(Path(workdir) / ("prog.exe" if os.name == "nt" else "prog"))
        subprocess.run([CXX, *FLAGS, "-o", exe, program], check=True)
        return [exe]
    return [sys.executable, program]


def run(command, text):
    r = subprocess.run(command, input=text, capture_output=True, text=True,
                       timeout=10)
    return r.stdout if r.returncode == 0 else f"<crash {r.returncode}>"


def normalize(text):
    return [line.rstrip() for line in text.rstrip().splitlines()]


def main():
    if len(sys.argv) < 4:
        sys.exit(__doc__)
    gen, brute, program = sys.argv[1:4]
    rounds = int(sys.argv[4]) if len(sys.argv) > 4 else 500
    with tempfile.TemporaryDirectory() as work:
        fast = command_for(program, work)
        for seed in range(1, rounds + 1):
            test = run([sys.executable, gen, str(seed)], "")
            want = run([sys.executable, brute], test)
            got = run(fast, test)
```

Listing 2.6 (Continued) *stress.py: runs a generator, a brute force and a program on many seeds, and prints the first input on which the answers differ.*

```
            if normalize(got) != normalize(want):
                print(f"round {seed}: DIFFERENT")
                print("input:\n" + test.rstrip())
                print("expected:\n" + want.rstrip())
                print("got:\n" + got.rstrip())
                return
    print(f"{rounds} rounds: all answers equal")


if __name__ == "__main__":
    main()
```

A good generator for ADDREV must produce the inputs that matter. Random numbers from a uniform distribution rarely end in zero, so **Listing 2.7** deliberately makes zeros common: the digit string `"0000123456789"` contains four zeros among thirteen characters. The first digit is

never zero, because the input numbers are positive and have no leading zeros; the generator must respect the constraints, or the stress test reports "bugs" on inputs that can never occur.

Listing 2.7 `gen_addrev.py`: small random ADDREV inputs, rich in zeros.

```
"""gen_addrev.py SEED: a small random ADDREV input, rich in zeros."""
import random
import sys


def number(rng):
    digits = rng.randint(1, 6)
    first = str(rng.randint(1, 9))  # a positive number has no leading zero
    rest = "".join(rng.choice("0000123456789") for _ in range(digits - 1))
    return first + rest


def main():
    rng = random.Random(int(sys.argv[1]))
    cases = rng.randint(1, 5)
    print(cases)
    for _ in range(cases):
        print(number(rng), number(rng))


if __name__ == "__main__":
    main()
```

The brute force, **Listing 2.8**, follows the definition literally with strings and needs no reasoning about digits or arithmetic.

Listing 2.8 `brute_addrev.py`: ADDREV straight from the definition.

```
"""brute_addrev.py: ADDREV the slow, obvious way, with strings only."""
import sys


def main():
    data = sys.stdin.read().split()
    for c in range(int(data[0])):
        a, b = data[1 + 2 * c], data[2 + 2 * c]
        total = int(a[::-1]) + int(b[::-1])  # int() drops leading zeros
        print(int(str(total)[::-1]))


if __name__ == "__main__":
    main()
```

Running the stress test on the correct programs and on the wrong one, from the `tests` folder:

```
$ python3 stress.py gen_addrev.py brute_addrev.py ../cpp/addrev.cpp
$ python3 stress.py gen_addrev.py brute_addrev.py ../python/addrev.py 200
$ python3 stress.py gen_addrev.py brute_addrev.py ../cpp/addrev_bug.cpp
```

```
500 rounds: all answers equal
200 rounds: all answers equal
round 1: DIFFERENT
input:
2
21049 8739
24 7
expected:
93301
94
got:
093301
94
```

The two correct programs agree with the brute force on 500 and 200 random inputs. The wrong one fails in the first round: $\text{rev}(21049) + \text{rev}(8739) = 94012 + 9378 = 103390$, whose reversal as a string is `093301`. A small, concrete input that shows the bug is worth more than an hour of staring at code.

> **WARNING.** A stress test is only as good as its generator and its brute force. A generator that never produces the difficult inputs, here numbers ending in zero, lets every bug through; a brute force that shares an idea with the fast program shares its bugs too. Write the brute force from the definition, as differently from the fast program as you can.

2.6 FCTRL2: Reading the Size of the Answer

The problem (SPOJ FCTRL2, "Small factorials", `https://www.spoj.com/problems/FCTRL2/`) [4]. The input begins with the number of test cases t, $1 \le t \le 100$; each case is one integer n with $1 \le n \le 100$. For each case, print $n!$, the product $1 \cdot 2 \cdot 3 \cdots n$, exactly, on its own line.

The title says *small* factorials, and the constraints say $n \le 100$. A reader in a hurry concludes that the problem is trivial. A reader following the checklist stops at question 5: does the answer fit in a `long long`? The number of decimal digits of a positive integer x is $\lfloor \log_{10} x \rfloor + 1$, and the logarithm of a product is the sum of the logarithms, which gives Equation (2.2).

$$\text{digits}(n!) = \left\lfloor \sum_{k=1}^{n} \log_{10} k \right\rfloor + 1 \tag{2.2}$$

For $n = 100$ the sum is about 157.97, so 100! has 158 digits. A `long long` holds at most 19 digits (Section 1.10): the largest factorial that fits is $20! = 2\,432\,902\,008\,176\,640\,000$, and $21!$ already overflows. So the "small" factorials are very large numbers, and in C++ we must do the multiplication ourselves, the way we learned to multiply at school.

The idea. Store the number as a list of decimal digits, least significant first: 120 is stored as `0, 2, 1`. To multiply it by k, go through the digits from the ones digit upward, multiply each by k, add the *carry* from the previous position, keep the last digit of the result in place and pass the rest on as the new carry. When the digits are exhausted, the remaining carry becomes new digits at the top.

Why does this never overflow an `int`? Each step computes $d \cdot k + c$ with a digit $d \le 9$ and $k \le 100$. The carry passed on is $\lfloor (d \cdot k + c)/10 \rfloor$; if $c \le 99$, then $d \cdot k + c \le 900 + 99 = 999$ and the next carry is at most 99 again. So by induction every value stays below 1000, far below `INT_MAX`.

Listing 2.9 `fctrl2.cpp`: FCTRL2 with schoolbook multiplication on a vector of decimal digits.

```cpp
// SPOJ FCTRL2 (Small factorials): n! for n <= 100, digit by digit.
#include <iostream>
#include <vector>

// digits[0] is the ones digit, digits[1] the tens digit, and so on.
std::vector<int> factorial(int n) {
    std::vector<int> digits = {1};
    for (int k = 2; k <= n; k++) {
        int carry = 0;
        for (int& d : digits) {
            int value = d * k + carry;  // at most 9 * 100 + carry
            d = value % 10;
            carry = value / 10;
        }
        while (carry > 0) {
            digits.push_back(carry % 10);
            carry /= 10;
        }
    }
    return digits;
}

int main() {
    int cases;
    std::cin >> cases;
    while (cases--) {
        int n;
        std::cin >> n;
        std::vector<int> digits = factorial(n);
        for (auto it = digits.rbegin(); it != digits.rend(); ++it) {
            std::cout << *it;
        }
        std::cout << '\n';
    }
    return 0;
}
```

Computing $n!$ takes $n - 1$ multiplications, each going over at most 158 digits, so a case costs at most about $100 \times 158 \approx 1.6 \times 10^4$ steps and all 100 cases at most 1.6×10^6: nothing, by the standards of Chapter 3. The digits are printed from the most significant one down, with a reverse iterator.

Why does the printed number never start with a zero? The vector starts as the single digit 1, and a new digit is appended only while the carry is positive, as its last digit `carry % 10` followed by the higher digits of the same carry. The very last digit appended is the leading digit of the last carry, which is not zero, because the loop stops as soon as the carry becomes zero. So the most significant stored digit is never 0, and printing from the top produces no leading zeros, which is exactly the property that ADDREV's buggy version lacked.

Listing 2.10 `fctrl2.py`: FCTRL2 in Python, whose integers have no fixed size.

```python
"""SPOJ FCTRL2 (Small factorials): n! for n <= 100; Python ints have no limit."""
import sys


def factorial(n):
    result = 1
    for k in range(2, n + 1):
        result *= k
    return result


def main():
    data = sys.stdin.read().split()
    cases = int(data[0])
    print("\n".join(str(factorial(int(x))) for x in data[1:1 + cases]))


if __name__ == "__main__":
    main()
```

In Python the problem really is small: integers grow as needed, and the obvious loop is the whole solution. (The standard library also has `math.factorial`; the loop is written out to show that nothing special happens.) This is one of the cases where the choice of language changes the difficulty of a problem, and it is worth remembering when a problem involves numbers beyond 64 bits: in a contest that allows Python, a big-number problem may be a five-minute problem in Python and a thirty-minute problem in C++.

```
$ printf '4\n1\n2\n5\n3\n' | ./cpp/fctrl2
```

```
1
2
120
6
```

The Python program prints the same four lines. For a stronger test, there is no need for random inputs: there are only 100 possible values of n, so we can test *all* of them. **Listing 2.11** feeds the numbers 1 to 100 to a program and compares every line with `math.factorial`.

Listing 2.11 `check_fctrl2.py`: checks a FCTRL2 program on every possible input.

```python
"""check_fctrl2.py PROGRAM: runs PROGRAM on n = 1..100; compares math.factorial."""
import math
import subprocess
import sys


def main():
    program = sys.argv[1]
    command = [sys.executable, program] if program.endswith(".py") else [program]
    text = "100\n" + "\n".join(str(n) for n in range(1, 101)) + "\n"
    got = subprocess.run(command, input=text, capture_output=True, text=True).stdout
    lines = got.split()
    bad = [n for n in range(1, 101) if lines[n - 1] != str(math.factorial(n))]
    print("all 100 factorials correct" if not bad else f"wrong for n = {bad}")
    print("digits of 100!:", len(lines[99]))


if __name__ == "__main__":
    main()
```

```
$ python3 check_fctrl2.py ../cpp/fctrl2
$ python3 check_fctrl2.py ../python/fctrl2.py
```

```
all 100 factorials correct
digits of 100!: 158
all 100 factorials correct
digits of 100!: 158
```

> **WARNING.** Python's integers have no size limit, but since Python 3.11 the conversion of an integer to a decimal string, as done by `str()` and `print()`, refuses numbers of more than 4300 digits and raises `ValueError`, a protection against denial-of-service attacks on web servers [5]. 100! with its 158 digits is far below the limit, but 2030! has 5835 digits, and a brute force that prints it fails. The call `sys.set_int_max_str_digits(0)` at the start of the program removes the limit; Chapter 3 needs it in one of its brute forces.

When the whole input space is small, test all of it; it is the strongest test there is. And a test of the largest case, here the 158-digit 100!, confirms Equation (2.2) independently.

2.7 ONP: Reading an Unfamiliar Rule

The problem (SPOJ ONP, "Transform the Expression", `https://www.spoj.com/problems/ONP/`) [6]. The input begins with the number of expressions, at most 100; each expression is written on one line, is at most 400 characters long, and uses the lower-case letters `a` to `z` as operands, the five binary operators `+ - * / ^`, and round brackets. Transform each expression into *Reverse Polish Notation* (RPN), in which every operator is written after its two operands, and print it. The statement gives the priorities of the operators from the lowest to the highest in the order `+`, `-`, `*`, `/`, `^`, and promises that every expression has exactly one RPN form, so that chains such as `a*b*c`, whose grouping would be ambiguous, do not occur.

Reading the rule that is not the usual one. Every reader knows that multiplication binds tighter than addition, and every reader knows that `+` and `-` have the *same* priority. The statement says otherwise: here `-` binds tighter than `+`, and `/` tighter than `*`. On the sample, which is fully bracketed, the difference never shows, and a program with school priorities passes the sample and fails the hidden tests. Take the expression `a+b*c-d`. With ONP's priorities, `*` binds first, then `-`, then `+`: $a + ((b \cdot c) - d)$, whose RPN is `abc*d-+`. With school priorities it would be $(a + b \cdot c) - d$, with RPN `abc*+d-`. Both are "correct"; only one is correct *here*. This is question 1 of the checklist in its purest form: read the rule that is written, not the rule you know.

What RPN is. In RPN the expression `a+b` is written `ab+`: first the operands, then the operator. Brackets are never needed, because the order of the symbols fixes the order of the operations. The sample gives three expressions: `(a+(b*c))` becomes `abc*+`, `((a+b)*(z+x))` becomes `ab+zx+*`, and `((a+t)*((b+(a+c))^(c+d)))` becomes `at+bac++cd+^*`.

The idea: a stack of waiting operators. Read the expression from left to right. Letters go straight to the output, because in RPN operands keep their order. An operator cannot be written yet, because its right operand has not been read; it waits on a *stack*, a pile where the last item put on is the first taken off. When a new operator arrives, every waiting operator whose priority is at least as high must be finished first, so it is popped and written; then the new operator waits. An opening bracket waits on the stack as a barrier; a closing bracket pops and writes operators down to the matching opening bracket. At the end, all remaining operators are written. This procedure is Dijkstra's *shunting-yard algorithm*, named after a railway yard where wagons wait on a side track [7].

Why is this correct? Two facts carry the argument. First, the letters come out in their original order, and so they must in RPN, because RPN never reorders operands. Second, an operator is

written exactly when both its operands are complete in the output. Its left operand was complete when the operator arrived. Its right operand is complete as soon as the algorithm meets a symbol that cannot belong to it: an operator of lower or equal priority, which ends the right operand of everything that binds at least as tightly; a closing bracket, which ends everything inside the bracket; or the end of the input, which ends everything. These are exactly the moments at which the algorithm pops. The operators on the stack always have strictly increasing priorities from bottom to top between two brackets, so popping from the top writes the tightest-binding operator, whose operands are the innermost, first, which is the order RPN requires. The promise of the statement that no chain like `a*b*c` occurs means that the choice between "pop on equal priority" and "do not pop" never changes the result; we pop, which is the usual rule for left-associative operators.

A trace of `a+b*c-d` shows the stack at work; the stack is written with its top on the right.

Table 2.2 The shunting-yard algorithm on `a+b*c-d` with ONP's priorities.

Read	Action	Output so far	Stack
`a`	write	`a`	
`+`	stack empty: push	`a`	`+`
`b`	write	`ab`	`+`
`*`	`+` has lower priority: push	`ab`	`+ *`
`c`	write	`abc`	`+ *`
`-`	pop `*` (priority 3 ≥ 2), stop at `+` (1 < 2), push	`abc*`	`+ -`
`d`	write	`abc*d`	`+ -`
end	pop everything	`abc*d-+`	

Listing 2.12 `onp.cpp`: ONP with the shunting-yard algorithm; the priorities are those of the statement.

```cpp
// SPOJ ONP (Transform the Expression): infix to Reverse Polish Notation.
#include <iostream>
#include <string>

int priority(char op) {
    switch (op) {
        case '+': return 1;
        case '-': return 2;
        case '*': return 3;
        case '/': return 4;
        case '^': return 5;
    }
    return 0;  // '(' stays on the stack until its ')'
}

std::string toRpn(const std::string& expr) {
    std::string out, ops;  // ops is used as a stack of operators
    for (char ch : expr) {
        if (ch >= 'a' && ch <= 'z') {
            out += ch;
        } else if (ch == '(') {
            ops += ch;
        } else if (ch == ')') {
            while (!ops.empty() && ops.back() != '(') {
                out += ops.back();
                ops.pop_back();
            }
            if (!ops.empty()) ops.pop_back();  // drop the '('
        } else {
            while (!ops.empty() && priority(ops.back()) >= priority(ch)) {
                out += ops.back();
                ops.pop_back();
            }
            ops += ch;
        }
    }
    while (!ops.empty()) {
        out += ops.back();
        ops.pop_back();
    }
    return out;
}
```

Listing 2.12 (Continued) *onp.cpp: ONP with the shunting-yard algorithm; the priorities are those of the statement.*

```cpp
int main() {
    int cases;
    std::cin >> cases;
    while (cases--) {
        std::string expr;
        std::cin >> expr;
        std::cout << toRpn(expr) << '\n';
    }
    return 0;
}
```

Listing 2.12 is the algorithm in C++, and **Listing 2.13** in Python. The function `priority` encodes the statement's list, with 0 for the opening bracket, so that the popping loop for an operator stops at a bracket. The stack is a `std::string` used only at its end: `back()` looks at the top, `+=` pushes and `pop_back()` pops. Each character is pushed and popped at most once, so the conversion is linear in the length of the expression.

Listing 2.13 `onp.py`: the same algorithm in Python, with a list as the stack.

```python
"""SPOJ ONP (Transform the Expression): infix to Reverse Polish Notation."""
import sys

PRIORITY = {"+": 1, "-": 2, "*": 3, "/": 4, "^": 5, "(": 0}


def to_rpn(expr):
    out, ops = [], []  # ops is used as a stack of operators
    for ch in expr:
        if "a" <= ch <= "z":
            out.append(ch)
        elif ch == "(":
            ops.append(ch)
        elif ch == ")":
            while ops and ops[-1] != "(":
                out.append(ops.pop())
            if ops:
                ops.pop()  # drop the "("
        else:
            while ops and PRIORITY[ops[-1]] >= PRIORITY[ch]:
                out.append(ops.pop())
            ops.append(ch)
    while ops:
        out.append(ops.pop())
    return "".join(out)


def main():
    data = sys.stdin.read().split()
    cases = int(data[0])
    print("\n".join(to_rpn(e) for e in data[1:1 + cases]))


if __name__ == "__main__":
    main()
```

```
$ printf '3\n(a+(b*c))\n((a+b)*(z+x))\n((a+t)*((b+(a+c))^(c+d)))\n' | ./cpp/onp
$ printf '1\na+b*c-d\n' | ./cpp/onp
```

```
abc*+
ab+zx+*
at+bac++cd+^*
abc*d-+
```

The Python program prints the same lines. The fourth line is the one the sample does not test, and it matches the hand trace of **Table 2.2**.

Testing it. A brute force for ONP must be *different* from the stack method. **Listing 2.14** uses *recursive descent*: an expression at priority level ℓ is a sequence of expressions at level $\ell + 1$ joined by the operator of level ℓ, and at the highest level an expression is either a letter or a bracketed expression. The grammar is written out directly, one level per operator in the order of the statement, so its correctness can be checked by reading it.

Listing 2.14 `brute_onp.py`: ONP by recursive descent, one grammar level per priority.

```
"""brute_onp.py: ONP by recursive descent, one grammar rule per priority level."""
import sys

LEVELS = "+-*/^"


def parse(expr, pos, level):
    """Parse operators of priority level or higher; return (rpn, next position)."""
    if level == len(LEVELS):
        if expr[pos] == "(":
            rpn, pos = parse(expr, pos + 1, 0)
            return rpn, pos + 1  # skip ")"
        return expr[pos], pos + 1
    left, pos = parse(expr, pos, level + 1)
    while pos < len(expr) and expr[pos] == LEVELS[level]:
        right, pos = parse(expr, pos + 1, level + 1)
        left = left + right + LEVELS[level]
    return left, pos


def main():
    data = sys.stdin.read().split()
    for expr in data[1:1 + int(data[0])]:
        print(parse(expr, 0, 0)[0])


if __name__ == "__main__":
    main()
```

The generator, **Listing 2.15**, builds a random expression tree and prints it with brackets exactly where the priorities require them (around a child whose operator does not bind tighter than its parent's), and sometimes with extra brackets. It never produces an unbracketed chain of equal priority, which the statement excludes.

Listing 2.15 `gen_onp.py`: random ONP expressions with minimal and extra brackets.

```python
"""gen_onp.py SEED: random ONP expressions, with brackets where they are needed."""
import random
import sys

OPS = "+-*/^"
PRIORITY = {op: i for i, op in enumerate(OPS, start=1)}


def build(rng, depth):
    """Return (text, priority of its top operator; 9 for a single letter)."""
    if depth == 0 or rng.random() < 0.25:
        return rng.choice("abcdefghijklmnopqrstuvwxyz"), 9
    op = rng.choice(OPS)
    parts = []
    for _ in range(2):
        text, prio = build(rng, depth - 1)
        # bracket a child whose operator does not bind tighter, or at random
        if prio <= PRIORITY[op] or (prio < 9 and rng.random() < 0.3):
            text = "(" + text + ")"
        parts.append(text)
    return parts[0] + op + parts[1], PRIORITY[op]


def main():
    rng = random.Random(int(sys.argv[1]))
    cases = rng.randint(1, 4)
    print(cases)
    for _ in range(cases):
        text, _ = build(rng, rng.randint(1, 5))
        print(text if rng.random() < 0.5 else "(" + text + ")")


if __name__ == "__main__":
    main()
```

```
$ python3 gen_onp.py 3
```

```
2
((((g-w)/p^m)+(v+y)^y)/y)*((o^e/e)/(y/u*s/l))/(((v^w)+(s^s))-(c+p)^(c/l))
((m+w)+(i^q))/a
```

```
$ python3 stress.py gen_onp.py brute_onp.py ../cpp/onp.cpp
$ python3 stress.py gen_onp.py brute_onp.py ../python/onp.py 200
```

```
500 rounds: all answers equal
200 rounds: all answers equal
```

The generated expressions are far less friendly than the sample: they mix brackets and priorities, and rely on `-` binding tighter than `+`. Both programs agree with the brute force on every one of them.

2.8 Using It for Real

Write the restatement in your notebook. For every problem, before coding, write the restatement, the answers to **Table 2.1**, and the hand calculation of the sample. It takes three minutes. After a few weeks you will do much of it in your head, but on a hard problem, or late at night, write it down again.

Keep a stress kit. The three files `stress.py`, a generator and a brute force are reusable for any problem with a single correct answer. Copy `stress.py` into the folder of each new problem, and write the generator and the brute force before, or right after, the real solution. A brute force for a problem is often a few lines: try all possibilities, compute from the definition, or use a big-number library. When the real solution gets WA on the judge, you are ready.

Make the generator mean. Small inputs find more bugs than large ones, because a bug that shows on a tiny input is easy to understand, and small inputs hit edge cases more often. Bias the generator toward the edges the constraints allow: the smallest n, equal values, zeros, the largest values.

Statements in a foreign language. Almost every SPOJ statement is written in English, some in Polish or another language, and many readers of this book read English as a second language. That is no disadvantage if you use it well: a reader who cannot skim is forced to read every sentence, which is exactly what the checklist asks for. Translate the task sentence, the input format, the output format and the constraints word by word into your own language, in writing, before you start; the story can be skimmed. Look up every word whose meaning you are not sure of, and pay special attention to the small words that carry rules: *at most*, *at least*, *exactly*, *distinct*, *non-negative*, *inclusive*, *respectively*, *consecutive*, *separated by*. "Non-negative" includes zero; "positive" does not. "Between 1 and n inclusive" includes both ends. "The numbers are distinct" means you may rely on there being no duplicates. A machine translation of a whole statement is useful as a first overview, but it often loses exactly these small words, so check the rules against the original.

Here is a restatement card for ADDREV of Section 2.5, of the kind you should write for every problem:

```
ADDREV  count first (N cases)  1 line per case: a b (reversed numbers, positive)
task:   print rev(rev(a) + rev(b)), rev drops leading zeros
edges:  sum ending in 0 (e.g. 305 794 -> 1); single digits; 1-digit + many-digit
sizes:  small numbers, sum has one more digit -> long long
output: one number per line, no leading zeros
sample: 24 1 -> 34; 4358 754 -> 1998; 305 794 -> 1 (checked by hand)
```

Reading specifications at work. The checklist transfers directly to the documents a developer reads: an interface description of a payment service, a file format prescribed by a tax office, a data exchange agreed with a hospital system. Each has an input format, an output format, limits, special cases stated once, and examples that cover only the easy paths. The habit of writing a restatement and asking "what happens at zero, at the maximum, with duplicates?" is exactly what prevents the expensive bugs in such work.

2.9 Common Mistakes, and How to Avoid Them

- **Skimming the output format.** Missing empty lines, extra spaces, `Case #1:` versus `Case 1:`: all are WA. Read the output format word by word, and compare your sample output character by character, or with `minijudge.py`.
- **Assuming school rules.** ONP's `-` binds tighter than `+`. When a statement defines something, even something you "know", use its definition.
- **Looping on `eof()`.** `while (!std::cin.eof())` runs one time too many. Put the read in the loop condition.
- **Reading lines when tokens would do.** Empty lines, missing final line breaks and Windows line endings break line-based reading. Read tokens unless line structure matters.
- **Not checking the size of the answer.** 100! has 158 digits. Estimate the size of intermediate values with Equation (2.2)-style reasoning before choosing a type.
- **Keeping leading zeros.** A reversed number, a sum, a product: when the statement says "without leading zeros", strip them, and let the generator produce the inputs that create them.

- **A brute force that copies the fast solution.** If both use the same idea, a stress test cannot find a mistake in the idea. Write the brute force from the definition.
- **Losing the small words in translation.** "Non-negative" is not "positive", "at most" is not "exactly", "distinct" is a promise you may use. Translate the rules word by word and check a machine translation against the original.
- **A generator that breaks the constraints.** It reports differences on inputs that never occur, and you waste time on non-bugs. Every generated input must be legal.

2.10 Chapter Summary

- A problem statement has six zones: header with limits, story and task, input format, output format, constraints and samples (**Figure 2.1**). The constraints decide both the algorithm and the edge cases; the samples show only the format.
- An eight-question checklist (**Table 2.1**) and a written restatement catch most misreadings before any code exists.
- Judge input arrives in three shapes: count first, sentinel, and until end of file. The last is read with the read in the loop condition in C++ (**Listing 2.1**) and with `sys.stdin.read().split()` in Python (**Listing 2.2**); reading tokens rather than lines is robust against empty lines and line endings.
- ADDREV computes $\text{rev}(\text{rev}(a) + \text{rev}(b))$ (Equation (2.1)); the trap is the leading zeros of the reversed sum. A stress test (**Listing 2.6**) with a zero-rich generator and a brute force from the definition exposed the wrong version in its first round.
- FCTRL2 asks for numbers of up to 158 digits (Equation (2.2)), far beyond `long long`; C++ needs schoolbook multiplication on digits (**Listing 2.9**), Python needs nothing special (**Listing 2.10**). With only 100 possible inputs, testing all of them is the strongest test.
- ONP defines its own priorities, with `-` above `+`; the shunting-yard algorithm (**Table 2.2**) converts to RPN in linear time, and a recursive-descent brute force (**Listing 2.14**) confirms it on generated expressions.

2.11 Exercises

1. Write the restatement and answer the eight questions of **Table 2.1** for SPOJ TEST from Chapter 1.
2. Compute by hand, using Equation (2.1), the ADDREV answer for the pairs (1,1), (90,10) and (5,5). Which of them does `addrev_bug.cpp` get wrong?
3. Use Equation (2.2) to find the number of digits of 25! without computing it, and check the result with the programs of Section 2.6.
4. Convert `a-b+c^d/e*f` to RPN by hand with ONP's priorities, using a trace like **Table 2.2**, and check your answer with `onp.py`.
5. A problem says: "The input contains several test cases. Each test case is a line with two integers n and k. The input ends with a line containing `0 0`, which must not be processed." Write the input loop in C++ and in Python.
6. Change `gen_addrev.py` so that it produces digits uniformly from `0123456789`, and run the stress test on `addrev_bug.cpp` again. In which round does it fail now? What does that tell you about generators?

2.12 Solutions to the Exercises

1. *Restatement:* the input is a sequence of integers of at most two digits, one per line; print them one per line until the first 42, printing neither the 42 nor anything after it. (1) Echo the numbers before the first 42. (2) No count; the input ends at the sentinel 42 or at end of file, and numbers may follow the 42. (3) The number of lines is not bounded, so the program must be linear and must not store the numbers in a fixed-size array. (4) The first number can be 42, so the output can be empty; numbers can be 0. (5) At most two digits: `int` is plenty. (6) One number per line. (7) The answer is unique. (8) The sample 1, 2, 88, 42, 99 gives 1, 2, 88.

2. (1,1): $\text{rev}(1) + \text{rev}(1) = 2$, reversed 2. (90,10): $\text{rev}(90) = 9$ and $\text{rev}(10) = 1$, so the sum is 10, and $\text{rev}(10) = 1$. (5,5): the sum is 10 again, answer 1. `addrev_bug.cpp` prints `2`, `01` and `01`: it is wrong on the last two, where the sum ends in zero. (Strictly, 90 and 10 cannot appear as input, because reversed numbers never end in zero; the second case is shown only for the arithmetic.)

3. $\sum_{k=1}^{25} \log_{10} k = \log_{10} 25! \approx 25.19$, so $25!$ has $\lfloor 25.19 \rfloor + 1 = 26$ digits. The programs print `15511210043330985984000000`, which has 26 digits.

4. The trace: `a` out; `-` push; `b` out; `+` arrives: `-` has priority $2 \geq 1$, pop it, output `ab-`, push `+`; `c` out; `^` push; `d` out; `/` arrives: pop `^` ($5 \geq 4$), stop at `+`, push `/`; `e` out; `*` arrives: pop `/` ($4 \geq 3$), stop at `+`, push `*`; `f` out; end: pop `*`, then `+`. The result is `ab-cd^e/f*+`, and `onp.py` prints the same.

5. In C++, the read and the sentinel test go in the loop condition:

```
long long n, k;
while (std::cin >> n >> k && !(n == 0 && k == 0)) {
    // process one test case
}
```

In Python, with all tokens read at once:

```
data = sys.stdin.read().split()
i = 0
while i + 1 < len(data):
    n, k = int(data[i]), int(data[i + 1])
    i += 2
    if n == 0 and k == 0:
        break
    # process one test case
```

Both loops also stop at end of file, in case a file lacks the closing `0 0`.

6. With uniform digits, a number ends in zero with probability 1/10 and a sum ends in zero only occasionally, so the bug needs more rounds to show: with uniform digits the stress test reports the first difference only in round 5 instead of round 1. On this problem both generators find the bug quickly, but on a problem where the special case is rarer, a uniform generator can run thousands of rounds without hitting it. The lesson: a generator should be biased toward the inputs that are special for the problem, not uniform.

References

[1] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Singapore: Lulu, 2020.

[2] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

[3] SPOJ, "ADDREV — Adding Reversed Numbers." [Online]. Available: https://www.spoj.com/problems/ADDREV/ (accessed Sep. 2026).

[4] SPOJ, "FCTRL2 — Small factorials." [Online]. Available: https://www.spoj.com/problems/FCTRL2/ (accessed Sep. 2026).

[5] Python Software Foundation, "Integer string conversion length limitation," *Python 3.14 Documentation: Built-in Types*. [Online]. Available: https://docs.python.org/3/library/stdtypes.html#int-max-str-digits (accessed Sep. 2026).

[6] SPOJ, "ONP — Transform the Expression." [Online]. Available: https://www.spoj.com/problems/ONP/ (accessed Sep. 2026).

[7] E. W. Dijkstra, "Algol 60 translation: An Algol 60 translator for the X1 and making a translator for Algol 60," Mathematisch Centrum, Amsterdam, Report MR 35/61, 1961.

Chapter 3

Picking the Right Algorithm for the Constraints

A program can be completely correct and still be rejected, because it is too slow. The judge gives it a second or two, and a correct method that needs an hour is, for the judge, a wrong method. The good news is that you can usually tell in advance, before writing a line, whether a method is fast enough: the constraints of the problem and a little arithmetic are enough. This chapter teaches that arithmetic. By the end of it you will be able to measure how many simple steps your computer makes per second in C++ and in Python; turn a time limit and the largest input into a budget of steps; estimate the steps an algorithm needs by counting its loops, including the loop over test cases; read off from a table which complexity classes fit which input sizes; and solve three real SPOJ problems whose obvious solutions are far too slow, FCTRL, LASTDIG and PRIME1, with a counting formula, repeated squaring and a segmented sieve, each checked against a brute force and, for FCTRL, watched as the slow version runs into the time limit of our own judge.

3.1 The Queue at the Photocopy Shop

Near every Indonesian campus there is a photocopy shop, and at the start of every semester there is a queue in front of it. A student asks for a single copy of a two-page form: the operator lays each page on the glass and presses the button, and it is done in a minute. The next student wants forty copies of a two-hundred-page reader. Laying pages on the glass one by one would take the whole afternoon, so the operator puts the whole stack into the automatic feeder, sets the counter to forty, and goes to serve the next customer. Nobody taught the operator complexity theory. The operator simply knows that the method that is best for two pages is hopeless for eight thousand, and chooses the method *after* hearing the size of the job.

Competitive programming asks for the same judgment, and the constraints of a problem are the size of the job. Rina's team met this lesson in the problem FCTRL, which asks how many zeros stand at the end of $n!$. Their first program was obviously correct: it went through all the numbers from 1 to n and counted factors of five. It gave the right answer on the sample in an instant. On the judge it received *time limit exceeded*, because n can be a billion and there can be a hundred thousand of them. Section 3.4 shows the two-line formula that replaces the loop, and runs both programs through our judge from Chapter 1, where the difference is not a few percent but a factor of more than a million.

Who benefits from learning to estimate running time? Everyone who writes programs that meet real data. A report that works on a test database of a hundred customers and freezes on the real one with two million; a search that is fine in the demo and too slow in production; a script that processes one day of sales in a second and a year of sales in six hours. The arithmetic of this chapter predicts all of these before they happen, which is much cheaper than discovering them afterwards.

3.2 From a Time Limit to a Budget of Steps

Chapter 1 gave the basic estimate in Section 1.3: the running time is the number of simple steps divided by the steps per second. Turned around, it gives the *budget*: the number of steps that fit in the time limit.

$$\text{budget} = \text{time limit} \times \text{steps per second} \tag{3.1}$$

The time limit is printed on the problem page. The steps per second must be measured, because it depends on the machine, the language and what a "step" is. **Listing 3.1** measures it in C++: it runs a loop of 2×10^8 rounds, each of which does a few shifts, exclusive-ors and an addition, the kind of cheap work that most inner loops do, and it times the sorting of ten million numbers.

Listing 3.1 `ops_budget.cpp`: measures simple loop steps per second and the time to sort 10^7 numbers in C++.

```
// ops_budget.cpp: how many simple steps does this computer make in one second?
#include <algorithm>
#include <chrono>
#include <cstdio>
#include <vector>

double secondsSince(std::chrono::steady_clock::time_point start) {
    std::chrono::duration<double> d = std::chrono::steady_clock::now() - start;
    return d.count();
}

int main() {
    const long long steps = 200000000;  // 2 * 10^8
    auto start = std::chrono::steady_clock::now();
    unsigned long long x = 88172645463325252ULL, check = 0;
    for (long long i = 0; i < steps; i++) {
        x ^= x << 13;  // a few cheap operations the compiler cannot skip
        x ^= x >> 7;
        x ^= x << 17;
        check += x & 1023;
    }
    double t = secondsSince(start);
    std::printf("simple loop: %lld steps in %.2f s = %.1e steps/s (check %llu)\n",
                steps, t, steps / t, check % 1000);

    const int n = 10000000;  // sort 10^7 numbers: about n log2 n = 2.3 * 10^8
    std::vector<unsigned> a(n);
    for (int i = 0; i < n; i++) {
        x ^= x << 13;
        x ^= x >> 7;
        x ^= x << 17;
        a[i] = static_cast<unsigned>(x);
    }
    start = std::chrono::steady_clock::now();
    std::sort(a.begin(), a.end());
    std::printf("sort of %d numbers: %.2f s\n", n, secondsSince(start));
    return 0;
}
```

The value `check` is printed so that the compiler cannot decide that the loop has no visible effect and delete it, which optimizing compilers are allowed to do. The class `std::chrono::steady_clock` is a clock that never jumps, the right one for measuring durations. **Listing 3.2** does the same in Python, with ten times fewer rounds, because Python is much slower per step, and one million numbers to sort.

Listing 3.2 `ops_budget.py`: the same measurement in Python.

```
"""ops_budget.py: how many simple steps does Python make in one second?"""
import random
import time


def main():
    steps = 10_000_000  # 10^7: Python is far slower per step than C++
    start = time.perf_counter()
    x, check = 88172645463325252, 0
    mask = (1 << 64) - 1
    for _ in range(steps):
        x ^= (x << 13) & mask
        x ^= x >> 7
        x ^= (x << 17) & mask
        check += x & 1023
    t = time.perf_counter() - start
    print(f"simple loop: {steps} steps in {t:.2f} s = {steps / t:.1e} steps/s"
          f" (check {check % 1000})")

    n = 1_000_000
    a = [random.getrandbits(32) for _ in range(n)]
    start = time.perf_counter()
    a.sort()
    print(f"sort of {n} numbers: {time.perf_counter() - start:.2f} s")


if __name__ == "__main__":
    main()
```

On the machine used for this book, a shared Linux server that was busy with other work at the time:

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o cpp/ops_budget cpp/ops_budget.cpp
$ ./cpp/ops_budget
$ python3 python/ops_budget.py
```

```
simple loop: 200000000 steps in 0.37 s = 5.3e+08 steps/s (check 719)
sort of 10000000 numbers: 0.83 s
simple loop: 10000000 steps in 2.33 s = 4.3e+06 steps/s (check 739)
sort of 1000000 numbers: 0.30 s
```

The first two lines come from C++, the last two from Python. Your numbers will differ, perhaps by a factor of two in either direction, but the pattern will not. Compiled C++ managed about 5×10^8 loop rounds per second, each round being four or five machine operations. Python managed about 4×10^6 of the same rounds, more than a hundred times fewer: every Python operation goes through the interpreter, which looks up the types of the values, allocates new integer objects, and checks for overflow into big numbers. Sorting, on the other hand, is fast in both languages, because Python's `sort` is written in C: a million numbers in 0.3 seconds. The lesson for Python programmers is to let the built-in functions, written in C, do the heavy loops wherever possible.

For planning, round the measurements to powers of ten. A judge machine may be older than yours, and a step in a real program may be more expensive than our cheap loop round, so the working rule is:

> **KEY INSIGHT.** Plan with about 10^8 simple steps per second in C++ and about 10^6 to 10^7 in Python. A method whose step count, at the largest input, is below the budget of Equation (3.1) by a factor of ten is safe; one within a factor of three is risky; one above it will not pass, however well it is written.

3.3 Counting the Steps of an Algorithm

To compare a method with the budget we need its step count at the largest input, and for that we do not need exact counts, only the order of magnitude. The standard tool is the *big-O* notation [1]. Guides written for competitive programmers use exactly this kind of estimate as their first step on every problem [2], [3].

> **DEFINITION.** An algorithm runs in time $O(f(n))$ if there are constants c and n_0 such that on every input of size $n \geq n_0$ it performs at most $c \cdot f(n)$ steps. Informally: for large inputs, its step count grows at most like $f(n)$, up to a constant factor.

In practice, three rules give the order of almost every program in this book.

1. **A loop multiplies.** A loop that runs n times around a body of cost g costs $n \cdot g$. Two nested loops over n values cost $n \cdot n = n^2$.
2. **Halving gives a logarithm.** A loop that halves its range each round, as binary search does, or doubles a value until it exceeds n, runs about $\log_2 n$ times. $\log_2 10^9 \approx 30$: a billion becomes one after thirty halvings.
3. **Test cases add up.** If there are T test cases and each costs $f(n)$, the whole input costs $T \cdot f(n)$. This rule is the one beginners forget: a method that is fast for one case can be hopeless for a hundred thousand.

$$\text{total steps} \approx T \cdot f(n_{\max}) \tag{3.2}$$

Equation (3.2), compared with Equation (3.1), is the whole method. Some problems state instead a bound on the *sum* of the sizes of all test cases; then the total is bounded by f of that sum, which is usually far smaller than $T \cdot f(n_{\max})$.

A worked count shows the rules at work. Suppose a problem gives $T \leq 10$ arrays of $n \leq 10^4$ numbers and asks, for each array, how many pairs of positions $i < j$ hold equal values. The obvious program has two nested loops: i runs from 0 to $n-1$ and, for each i, j runs from $i+1$ to $n-1$. The inner loop runs $n-1$ times for $i = 0$, $n-2$ times for $i = 1$, and so on down to 0, so the comparisons number

$$(n-1) + (n-2) + \cdots + 1 + 0 = \frac{n(n-1)}{2} \tag{3.3}$$

which for $n = 10^4$ is about 5×10^7. Times $T = 10$ gives 5×10^8 by Equation (3.2): five times the planning budget of 10^8, so the double loop is risky at best. Sorting each array first and counting runs of equal values costs $O(n \log n)$ per array, about $10 \cdot 10^4 \cdot 14 \approx 1.4 \times 10^6$ steps in all, a few hundred times less. Equation (3.3) is worth remembering: every "all pairs" loop costs about $n^2/2$, half of the square, which is still quadratic, since the constant $1/2$ does not change the growth.

Figure 3.1 shows how the common complexity classes grow, on logarithmic axes on which every power of n is a straight line. The two shaded bands are about one second of C++ and about one second of Python.

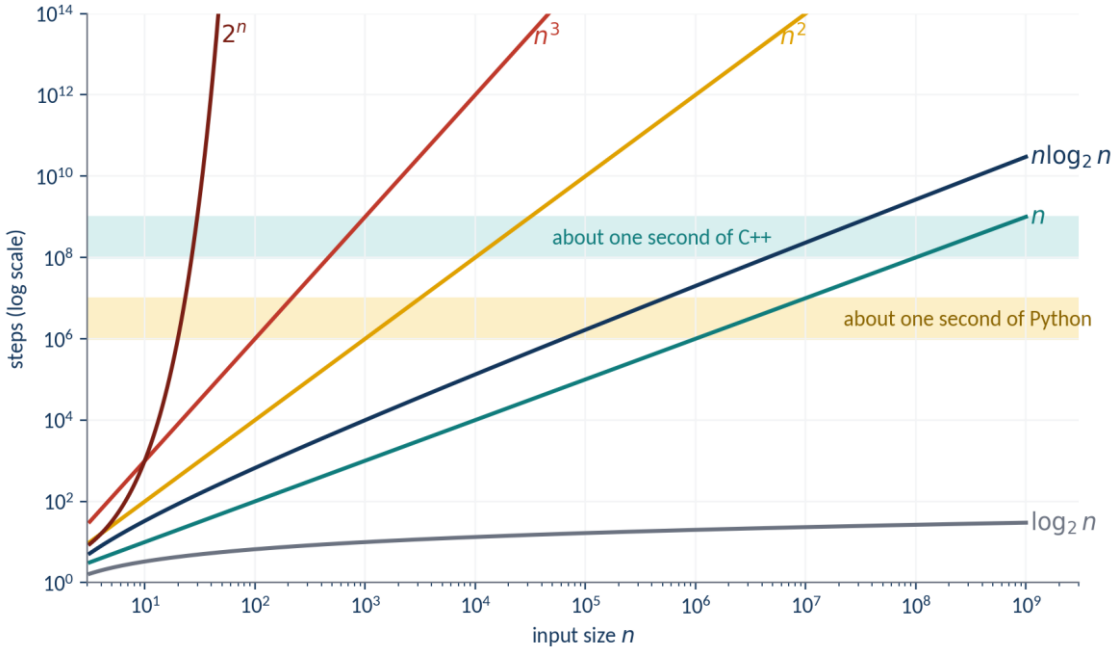


Figure 3.1 Steps needed by the common complexity classes as the input size grows, against about one second of C++ (teal band) and of Python (amber band). Where a curve crosses a band is the largest input that class can handle in a second.

Reading the figure from left to right gives the table that experienced competitors carry in their heads, **Table 3.1**. Find the row of the largest n in the constraints; the complexity next to it is the slowest that is likely to pass in C++.

Table 3.1 The complexity budget in C++: the largest input size in the constraints and the slowest complexity that fits in about one second. For Python, move one or two rows down.

Largest n	Slowest complexity that fits in about a second	Typical methods
up to 10–11	$O(n!)$	trying all orders
up to 20–25	$O(2^n)$, $O(2^n \cdot n)$	trying all subsets, bitmask dynamic programming
up to about 500	$O(n^3)$	triple loops, Floyd–Warshall, interval dynamic programming
up to about 5,000	$O(n^2)$	double loops, simple dynamic programming
up to about 10^6	$O(n\log n)$	sorting, binary search, heaps, segment trees
up to about 10^8	$O(n)$	one pass, prefix sums, two pointers, sieves
beyond 10^9	$O(\sqrt{n})$, $O(\log n)$, $O(1)$	formulas, repeated squaring, number theory

Two warnings about this table. First, it describes C++; for Python, the $O(n)$ row ends near 10^7 rather than 10^8, and the $O(n^2)$ row near 1,000 to 2,000. Second, near a row boundary, compute instead of guessing: $n = 10^5$ with an $O(n^2)$ method is 10^{10} steps, a hundred times the budget, while $n = 3000$ with the same method is 9×10^6, comfortably inside it. The jump between neighboring rows is large, and there is rarely a middle ground, so the table is a fast and reliable first filter.

The table also works backwards, and this is one of the most useful tricks in competitive programming: the constraints are a hint. A problem setter who allows $n \le 20$ is hinting at an exponential method over subsets; one who allows $n \le 10^5$ is ruling out anything quadratic; one who allows $n \le 10^{18}$ is saying that the answer comes from a formula or from $O(\log n)$ steps.

When you have no idea how to solve a problem, look at the constraints and ask which row they fall in.

3.4 FCTRL: When a Formula Replaces a Loop

The problem (SPOJ FCTRL, "Factorial", `https://www.spoj.com/problems/FCTRL/`) [4]. The input begins with the number of test cases T, about 10^5; each case is one integer n with $1 \le n \le 10^9$. For each case, print the number of zeros at the end of the decimal representation of $n!$. The story around it is about a mobile phone network, and it is pure decoration: the task is one sentence, the number of trailing zeros of $n!$.

Where the zeros come from. A trailing zero is a factor 10 of the number, and $10 = 2 \cdot 5$. In $n! = 1 \cdot 2 \cdots n$ there are always more factors 2 than factors 5 (every second number is even, only every fifth is a multiple of 5), so the number of trailing zeros equals the number of factors 5 in $n!$. For $n = 10$: the factors 5 come from 5 and 10, so $10! = 3\,628\,800$ has two trailing zeros.

The obvious method and its cost. Go through the multiples of 5 up to n and count how often each divides by 5. That is about $n/5$ rounds per case. With Equation (3.2): $T \cdot n/5 = 10^5 \cdot 2 \times 10^8 = 2 \times 10^{13}$ steps, which is 2×10^5 times the budget. **Table 3.1** says the same at a glance: $n = 10^9$ needs $O(\log n)$ or a formula. The obvious program, **Listing 3.3**, is correct, and we keep it as the brute force of our own tests, but it cannot pass.

Listing 3.3 `fctrl_naive.cpp`: FCTRL by counting the factors 5 of every multiple of 5, correct and far too slow.

```
// SPOJ FCTRL (Factorial), the slow way: count factors 5 in every multiple of 5.
#include <iostream>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int cases;
    std::cin >> cases;
    while (cases--) {
        long long n;
        std::cin >> n;
        long long zeros = 0;
        for (long long k = 5; k <= n; k += 5) {  // about n / 5 rounds per case
            long long m = k;
            while (m % 5 == 0) {
                zeros++;
                m /= 5;
            }
        }
        std::cout << zeros << '\n';
    }
    return 0;
}
```

The formula. Count the factors 5 differently: not number by number, but layer by layer. Every multiple of 5 up to n contributes at least one factor 5, and there are $\lfloor n/5 \rfloor$ of them. Every multiple of 25 contributes a second factor 5, and there are $\lfloor n/25 \rfloor$ of them. Every multiple of 125 contributes a third, and so on. A number that is divisible by exactly 5^j is counted once in each of the first j layers, which is exactly its number of factors 5. So the count is Equation (3.4), known as Legendre's formula [5].

$$Z(n) = \left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{25} \right\rfloor + \left\lfloor \frac{n}{125} \right\rfloor + \cdots = \sum_{j \geq 1} \left\lfloor \frac{n}{5^j} \right\rfloor \tag{3.4}$$

The sum is finite in practice, because $\lfloor n/5^j \rfloor = 0$ once $5^j > n$; for $n = 10^9$ that happens at $j = 13$, since $5^{13} \approx 1.2 \times 10^9$. So each case costs about 13 divisions, and all 10^5 cases about 1.3×10^6: a millionth of the obvious method. A worked example, for the sample value $n = 100$: $\lfloor 100/5 \rfloor = 20$, $\lfloor 100/25 \rfloor = 4$, $\lfloor 100/125 \rfloor = 0$, so $Z(100) = 24$.

Listing 3.4 implements it in C++, and **Listing 3.5** in Python.

Listing 3.4 `fctrl.cpp`: FCTRL with Legendre's formula, about 13 divisions per case.

```cpp
// SPOJ FCTRL (Factorial): trailing zeros of n! = n/5 + n/25 + n/125 + ...
#include <iostream>

long long trailingZeros(long long n) {
    long long zeros = 0;
    for (long long power = 5; power <= n; power *= 5) {
        zeros += n / power;  // multiples of 5, then of 25, then of 125, ...
    }
    return zeros;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int cases;
    std::cin >> cases;
    while (cases--) {
        long long n;
        std::cin >> n;
        std::cout << trailingZeros(n) << '\n';
    }
    return 0;
}
```

The variable `power` runs through 5,25,125, ...; it is a `long long` because the last power that the loop computes, the first one above n, can exceed the range of `int` when n is close to 10^9: $5^{13} = 1\,220\,703\,125 < 2^{31}$, but a careless bound such as $n \leq 2 \times 10^9$ would push it over. Using a 64-bit type for a value that is multiplied inside a loop is a cheap habit that prevents a whole class of bugs.

Listing 3.5 `fctrl.py`: the same formula in Python, reading all input at once.

```python
"""SPOJ FCTRL (Factorial): trailing zeros of n! = n//5 + n//25 + n//125 + ..."""
import sys


def trailing_zeros(n):
    zeros, power = 0, 5
    while power <= n:
        zeros += n // power  # multiples of 5, then of 25, then of 125, ...
        power *= 5
    return zeros


def main():
    data = sys.stdin.buffer.read().split()
    cases = int(data[0])
    sys.stdout.write("\n".join(str(trailing_zeros(int(x)))
                               for x in data[1:1 + cases]) + "\n")


if __name__ == "__main__":
    main()
```

The Python program reads all tokens at once with `sys.stdin.buffer.read().split()` and writes all answers with one call, which matters with 10^5 lines, as Chapter 4 shows. All three programs print the sample output:

```
$ printf '6\n3\n60\n100\n1024\n23456\n8735373\n' | ./cpp/fctrl
```

```
0
14
24
253
5861
2183837
```

Testing it. The brute force for FCTRL is the definition itself: compute $n!$ exactly and count its zeros, **Listing 3.6**. It is only practical for small n, so the generator, **Listing 3.7**, produces values up to 3000, with the edge values 1, 4, 5, 24, 25, 124 and 125, where a new power of 5 starts to count, appearing often.

Listing 3.6 `brute_fctrl.py`: FCTRL by computing $n!$ in full and counting its trailing zeros.

```python
"""brute_fctrl.py: FCTRL by computing n! in full and counting its zeros."""
import math
import sys


def main():
    sys.set_int_max_str_digits(0)  # allow str() of numbers with > 4300 digits
    data = sys.stdin.read().split()
    for x in data[1:1 + int(data[0])]:
        digits = str(math.factorial(int(x)))
        print(len(digits) - len(digits.rstrip("0")))


if __name__ == "__main__":
    main()
```

Listing 3.7 `gen_fctrl.py`: small FCTRL inputs with the boundary values of the powers of 5.

```python
"""gen_fctrl.py SEED: a small FCTRL input (n <= 3000, so brute force can check)."""
import random
import sys


def main():
    rng = random.Random(int(sys.argv[1]))
    cases = rng.randint(1, 8)
    print(cases)
    for _ in range(cases):
        print(rng.choice([1, 4, 5, 24, 25, 124, 125, rng.randint(1, 3000)]))


if __name__ == "__main__":
    main()
```

The brute force is the program that met the Python limit on integer-to-string conversion mentioned in Chapter 2: 2030! has more than 4300 digits, and without `sys.set_int_max_str_digits(0)` its `str()` fails. The stress test found that on its first round, which is another good reason to run the stress test on the brute force itself before trusting it.

```
$ cd tests
$ python3 stress.py gen_fctrl.py brute_fctrl.py ../cpp/fctrl.cpp 300
$ python3 stress.py gen_fctrl.py brute_fctrl.py ../python/fctrl.py 200
$ python3 stress.py gen_fctrl.py brute_fctrl.py ../cpp/fctrl_naive.cpp 200
```

```
300 rounds: all answers equal
200 rounds: all answers equal
200 rounds: all answers equal
```

All three agree with the brute force: the naive program is *correct*. Now the time. **Listing 3.8** writes two tests for our judge: the sample, and the largest legal input, 10^5 values of n between 9×10^8 and 10^9, with the expected answers computed by a third, independent implementation of Equation (3.4).

Listing 3.8 `make_fctrl_tests.py`: the sample and a maximum-size test for FCTRL, in the format of `minijudge.py`.

```
"""make_fctrl_tests.py: writes FCTRL/01 (the sample) and FCTRL/02 (the largest)."""
import random
from pathlib import Path

HERE = Path(__file__).resolve().parent / "FCTRL"


def zeros(n):
    count = 0
    while n:
        n //= 5
        count += n
    return count


def write(name, numbers):
    HERE.mkdir(exist_ok=True)
    text = f"{len(numbers)}\n" + "".join(f"{n}\n" for n in numbers)
    (HERE / f"{name}.in").write_text(text)
    (HERE / f"{name}.out").write_text("".join(f"{zeros(n)}\n" for n in numbers))


def main():
    rng = random.Random(5)
    write("01", [3, 60, 100, 1024, 23456, 8735373])
    write("02", [rng.randint(900_000_000, 1_000_000_000) for _ in range(100_000)])
    print("wrote FCTRL/01 and FCTRL/02")


if __name__ == "__main__":
    main()
```

With a time limit of one second, `minijudge.py` from Chapter 1 gives:

```
$ python3 make_fctrl_tests.py
$ python3 ../../Chapter01/python/minijudge.py FCTRL 1 ../cpp/fctrl_naive.cpp
$ python3 ../../Chapter01/python/minijudge.py FCTRL 1 ../cpp/fctrl.cpp
$ python3 ../../Chapter01/python/minijudge.py FCTRL 1 ../python/fctrl.py
```

```
wrote FCTRL/01 and FCTRL/02
  test 01: AC   0.01 s
  test 02: TLE  1.00 s
verdict: TLE (time limit exceeded)
  test 01: AC   0.00 s
  test 02: AC   0.03 s
verdict: AC (accepted)
  test 01: AC   0.03 s
  test 02: AC   0.22 s
verdict: AC (accepted)
```

Here is the verdict that Chapter 1 promised: a correct program, killed on the large test after one second, which is exactly what the judge does. By the estimate above it would need hours. The formula needs 0.03 seconds in C++ and 0.22 seconds in Python, most of which is reading the input and starting the interpreter. The Python version passes with a wide margin: when the algorithm is right, Python is often fast enough.

3.5 LASTDIG: Halving the Exponent

The problem (SPOJ LASTDIG, “The last digit”, `https://www.spoj.com/problems/LASTDIG/`) [6]. The input begins with the number of test cases, at most 30; each case gives two integers a and b with $0 \le a \le 20$ and $0 \le b \le 2\,147\,483\,000$, not both zero. Print the last decimal digit of a^b.

Reading the constraints. The base is tiny and the exponent is huge. a^b itself has up to about $b\log_{10}20 \approx 2.8 \times 10^9$ digits, so computing it is out of the question even in Python. The last digit of a product depends only on the last digits of the factors, so we can multiply modulo 10 all the way: $a^b \bmod 10$. Multiplying b times costs up to 2.1×10^9 steps per case, 6.4×10^{10} for 30 cases, far beyond the budget. **Table 3.1**, row "beyond 10^9", points to $O(\log b)$.

Repeated squaring. Write the exponent in binary. For $b = 10$, binary `1010`, $a^{10} = a^8 \cdot a^2$. The powers $a^1, a^2, a^4, a^8, \ldots$ are each the square of the previous one, so they cost one multiplication each, and we multiply into the result exactly those whose bit in b is 1:

$$a^b = \prod_{i\,:\,\text{bit } i \text{ of } b \text{ is } 1} a^{2^i} \qquad \text{with} \quad a^{2^{i+1}} = (a^{2^i})^2 \tag{3.5}$$

Equation (3.5) holds because $b = \sum_i \beta_i\, 2^i$ with bits $\beta_i \in \{0,1\}$, and $a^{x+y} = a^x a^y$. Since b has about $\log_2 b$ bits, at most 31 here, the method needs at most 31 squarings and 31 multiplications, all modulo 10. Every intermediate value is below 10, so the products are below 100 and nothing can overflow [1].

A worked example, $3^{10} \bmod 10$ from the sample: $b = 10 = 1010_2$. Start with result 1 and base 3.

1. Bit 0 of 10 is 0: result stays 1; base becomes $3^2 = 9$.
2. Bit 1 is 1: result $1 \cdot 9 = 9$; base becomes $9^2 = 81 \equiv 1$.
3. Bit 2 is 0: result stays 9; base becomes $1^2 = 1$.
4. Bit 3 is 1: result $9 \cdot 1 = 9$; base 1. No bits remain.

The last digit of $3^{10} = 59\,049$ is 9, as the method says.

Listing 3.9 and **Listing 3.10** implement the method.

Listing 3.9 `lastdig.cpp`: LASTDIG with repeated squaring modulo 10.

```
// SPOJ LASTDIG (The last digit): last digit of a^b by repeated squaring.
#include <iostream>

// (base ^ exp) % mod in about log2(exp) rounds.
long long powerMod(long long base, long long exp, long long mod) {
    long long result = 1 % mod;
    base %= mod;
    while (exp > 0) {
        if (exp & 1) result = result * base % mod;  // this bit of exp is 1
        base = base * base % mod;
        exp >>= 1;
    }
    return result;
}

int main() {
    int cases;
    std::cin >> cases;
    while (cases--) {
        long long a, b;
        std::cin >> a >> b;
        std::cout << powerMod(a, b, 10) << '\n';
    }
    return 0;
}
```

In `powerMod`, `exp & 1` is the lowest bit of the exponent, and `exp >>= 1` shifts it out, so the loop visits the bits from the lowest upward, exactly as in the worked example. The start value `1 % mod`

rather than `1` makes the function correct even for `mod = 1`, where every result is 0; it costs nothing and removes an edge case. The case $b = 0$ needs no special code: the loop does not run and the result is 1, the last digit of $a^0 = 1$.

Listing 3.10 `lastdig.py`: the same method in Python; the built-in `pow(a, b, 10)` does the same.

```
"""SPOJ LASTDIG (The last digit): last digit of a^b by repeated squaring."""
import sys


def power_mod(base, exp, mod):
    """(base ** exp) % mod in about log2(exp) rounds, like pow(base, exp, mod)."""
    result = 1 % mod
    base %= mod
    while exp > 0:
        if exp & 1:  # this bit of exp is 1
            result = result * base % mod
        base = base * base % mod
        exp >>= 1
    return result


def main():
    data = sys.stdin.read().split()
    cases = int(data[0])
    for c in range(cases):
        a, b = int(data[1 + 2 * c]), int(data[2 + 2 * c])
        print(power_mod(a, b, 10))


if __name__ == "__main__":
    main()
```

Python's built-in `pow` with three arguments computes a^b mod m by the same method, and in real code you should use it; the function is written out here to show the method, and the brute force uses the built-in, so that the stress test compares two independent implementations.

```
$ printf '2\n3 10\n6 2\n' | ./cpp/lastdig
```

```
9
6
```

```
$ python3 stress.py gen_lastdig.py brute_lastdig.py ../cpp/lastdig.cpp
$ python3 stress.py gen_lastdig.py brute_lastdig.py ../python/lastdig.py 200
```

```
500 rounds: all answers equal
200 rounds: all answers equal
```

The generator of this test, `gen_lastdig.py` in the `tests` folder, produces bases from 0 to 20 and exponents that are either small (0 to 5, where the edge cases live, including $b = 0$ and 0^b) or anywhere up to 2 147 483 000, and it never produces the excluded pair $a = b = 0$. Repeated squaring is one of the most reused ideas in this book. The same loop, with numbers replaced by matrices, computes the n-th Fibonacci number modulo m for n up to 10^{18} in about 60 matrix products; with a prime modulus p it computes the modular inverse a^{p-2} mod p; and it works for any operation that is associative, because only associativity is used when a^{x+y} is split into $a^x a^y$. Chapter 13 returns to it. There is also a completely different solution: the last digits of $a^1, a^2, a^3, \ldots$ repeat with a period that divides 4, so only b mod 4 matters; Exercise 4 explores it.

3.6 PRIME1: The Segmented Sieve

The problem (SPOJ PRIME1, "Prime Generator", `https://www.spoj.com/problems/PRIME1/`) [7]. The input begins with the number of test cases $t \le 10$; each case gives two numbers m and n with $1 \le m \le n \le 10^9$ and $n - m \le 10^5$. For each case, print every prime p with $m \le p \le n$, one per line, and separate the outputs of consecutive test cases by an empty line. This is the problem that Rina's team got wrong three times, in Chapter 2, because of that empty line; Section 2.2 of that chapter already read its constraints.

Reading the constraints. The numbers are large, up to 10^9, but each range is short, at most $10^5 + 1$ numbers, and there are at most 10 ranges: at most about 10^6 candidates in total. The shape pass adds: $m = 1$ is possible, and 1 is not prime; a range may contain a single number; and $p \cdot p$ for p up to about 31 623 is near 10^9, which fits in `int`, but a sum such as $m + p$ or a product computed a little carelessly can exceed it, so the program computes positions in `long long`.

The obvious methods and their cost. *Trial division* tests each candidate x by dividing it by $2, 3, \ldots, \lfloor\sqrt{x}\rfloor$, which is enough by Equation (3.6) below. A prime near 10^9 needs about 31 623 divisions; a range of 10^5 numbers near 10^9 contains about 4,800 primes, so ten ranges need about 1.5×10^9 divisions for the primes alone, more than ten times the budget, and division is one of the slowest machine operations. The ordinary *sieve of Eratosthenes* up to n would need an array of 10^9 entries, a gigabyte, and a pass over it for each case: both too slow and too large.

The idea. Two observations combine into the solution.

First, a composite number x always has a prime factor $p \le \sqrt{x}$. If $x = a \cdot b$ with $1 < a \le b$, then $a^2 \le a \cdot b = x$, so $a \le \sqrt{x}$, and any prime factor of a is also at most $\sqrt{x}$:

$$x \text{ composite} \Longrightarrow x \text{ has a prime factor } p \le \sqrt{x} \le \sqrt{n} \tag{3.6}$$

For $n \le 10^9$, $\sqrt{n} < 31\,623$, so the primes up to 31 623, of which there are 3,401, are enough to recognize every composite number in any range.

Second, instead of testing each number against the primes, let each prime *cross out* its multiples in the range, as the sieve of Eratosthenes does. For a prime p and a range $[m, n]$, the first multiple of p that is at least m is $\lceil m/p \rceil \cdot p$; from there, every p-th number is a multiple. We can also start no lower than p^2, because a smaller multiple $k \cdot p$ with $k < p$ has a smaller prime factor that already crossed it out, and starting at p^2 also protects p itself from being crossed out when it lies in the range. So each prime crosses out the positions

$$\max\left(p^2,\ \left\lceil \frac{m}{p} \right\rceil p\right),\ \ldots + p,\ \ldots + 2p,\ \ldots \le n \tag{3.7}$$

and the numbers that are never crossed out, except 1, are the primes of the range. This is the *segmented sieve*: an ordinary sieve once up to $\sqrt{n_{\max}}$, and then a small sieve over each range.

Its cost. Prime p crosses out about $(n - m)/p$ positions in a range, so a range costs about $(n - m) \sum_{p \le \sqrt{n}} 1/p$ steps. The sum of the reciprocals of the primes up to x grows like $\ln\ln x$, which for $x = 31\,623$ is about 2.6, so a range costs a few times 10^5 steps and ten ranges a few million, plus 3,401 start computations per range. Far inside the budget, in both languages.

Its memory. The time limit is not the only budget; the memory limit is another, and Chapter 6 treats it in detail. The segmented sieve needs one byte per number of the current range, at most

$10^5 + 1$ bytes, about 100 KB, plus the 3,401 base primes, about 14 KB as `int`s. An ordinary sieve up to 10^9 with one byte per number would need a gigabyte; even packed at one bit per number, as `std::vector<bool>` does, it would need 125 MB and a pass over a billion positions. So the segmented sieve wins on both budgets at once, which is typical: an algorithm that touches less data is usually both faster and smaller. Writing the byte count next to the step count, as a second line of the estimate, takes ten seconds and prevents the verdict that is reported as a runtime error but is really a memory limit.

A worked example on the range [10,30]: $\sqrt{30} < 6$, so the primes 2, 3 and 5 suffice. Prime 2 starts at $\max(4,10) = 10$ and crosses out 10, 12, ..., 30. Prime 3 starts at $\max(9, \lceil 10/3 \rceil \cdot 3) = 12$ and crosses out 12, 15, 18, 21, 24, 27, 30. Prime 5 starts at $\max(25,10) = 25$ and crosses out 25 and 30. The numbers left are 11, 13, 17, 19, 23, 29, which are exactly the primes between 10 and 30.

Listing 3.11 is the C++ program and **Listing 3.12** the Python one.

Listing 3.11 `prime1.cpp`: PRIME1 with a segmented sieve; the base primes are computed once, each range is sieved separately.

```cpp
// SPOJ PRIME1 (Prime Generator): a segmented sieve of Eratosthenes.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>

// All primes up to limit, by the ordinary sieve.
std::vector<int> smallPrimes(int limit) {
    std::vector<bool> composite(limit + 1, false);
    std::vector<int> primes;
    for (int p = 2; p <= limit; p++) {
        if (composite[p]) continue;
        primes.push_back(p);
        for (long long q = 1LL * p * p; q <= limit; q += p) composite[q] = true;
    }
    return primes;
}

// Append every prime in [m, n] to out, one per line.
void primesBetween(long long m, long long n, const std::vector<int>& base,
                   std::string& out) {
    std::vector<char> composite(n - m + 1, 0);  // composite[i] is about m + i
    for (int p : base) {
        long long pp = 1LL * p * p;
        if (pp > n) break;
        long long start = std::max(pp, (m + p - 1) / p * p);  // first multiple
        for (long long j = start; j <= n; j += p) composite[j - m] = 1;
    }
    for (long long x = std::max(m, 2LL); x <= n; x++) {
        if (!composite[x - m]) out += std::to_string(x) + '\n';
    }
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    const std::vector<int> base = smallPrimes(31623);  // 31623^2 > 10^9
    int cases;
    std::cin >> cases;
    std::string out;
    for (int c = 0; c < cases; c++) {
        long long m, n;
        std::cin >> m >> n;
        if (c > 0) out += '\n';  // an empty line between test cases
        primesBetween(m, n, base, out);
    }
```

Listing 3.11 (Continued) *prime1.cpp: PRIME1 with a segmented sieve; the base primes are computed once, each range is sieved separately.*

```cpp
    std::cout << out;
    return 0;
}
```

`smallPrimes` is the ordinary sieve up to 31 623, run once at the start. `primesBetween` allocates one byte per number of the range, `composite[i]` standing for the number $m + i$, and applies Equation (3.7) for each base prime; the loop over base primes stops as soon as $p^2 > n$, by Equation (3.6). The expression `(m + p - 1) / p * p` is the integer form of $\lceil m/p \rceil \cdot p$. The output of all cases is collected in one string and written at the end, with an empty line before every case except the first, which is exactly the rule of the output format: *between* cases, not after each.

Listing 3.12 `prime1.py`: the same segmented sieve in Python, crossing out with slice assignment.

```python
"""SPOJ PRIME1 (Prime Generator): a segmented sieve of Eratosthenes."""
import sys


def small_primes(limit):
    """All primes up to limit, by the ordinary sieve."""
    composite = bytearray(limit + 1)
    primes = []
    for p in range(2, limit + 1):
        if not composite[p]:
            primes.append(p)
            composite[p * p::p] = b"\x01" * len(range(p * p, limit + 1, p))
    return primes


def primes_between(m, n, base):
    """Every prime in [m, n]; composite[i] is about m + i."""
    size = n - m + 1
    composite = bytearray(size)
    for p in base:
        if p * p > n:
            break
        start = max(p * p, (m + p - 1) // p * p) - m  # first multiple, as index
        composite[start::p] = b"\x01" * len(range(start, size, p))
    return [m + i for i in range(size) if not composite[i] and m + i >= 2]


def main():
    data = sys.stdin.read().split()
    base = small_primes(31623)  # 31623 ** 2 > 10 ** 9
    cases = int(data[0])
    blocks = []
    for c in range(cases):
        m, n = int(data[1 + 2 * c]), int(data[2 + 2 * c])
        blocks.append("".join(f"{x}\n" for x in primes_between(m, n, base)))
    sys.stdout.write("\n".join(blocks))  # an empty line between test cases


if __name__ == "__main__":
    main()
```

The Python version must avoid a Python-level loop over the crossed-out positions, which would cost several million interpreted steps. It uses *slice assignment* instead: `composite[start::p] = b"\x01" * count` sets every p-th byte from `start` on in a single operation that runs inside the interpreter's C code. The right side must have exactly as many bytes as the slice has positions, which is `len(range(start, size, p))`, a number Python computes without building the range. The final list comprehension is the one remaining Python-level loop, once per number of the range.

```
$ printf '2\n1 10\n3 5\n' | ./cpp/prime1
```

```
2
3
5
7

3
5
```

This is the sample: the primes up to 10, an empty line, and the primes from 3 to 5. Notice that 1 is not printed, and that there is no empty line after the last case. The Python program prints the same lines.

Testing and timing. The brute force, `brute_prime1.py` in the `tests` folder, is trial division, fine for short ranges. The generator makes up to four ranges, either near 1, where the edge cases 1, 2 and 3 live, or just below 10^9, where the large numbers are, each at most a few hundred numbers wide so that trial division stays fast.

```
$ python3 stress.py gen_prime1.py brute_prime1.py ../cpp/prime1.cpp 300
$ python3 stress.py gen_prime1.py brute_prime1.py ../python/prime1.py 200
```

```
300 rounds: all answers equal
200 rounds: all answers equal
```

For the time, `make_prime1_worst.py` writes the largest input the constraints allow: ten ranges from 999 900 000 to 10^9. We time both programs with the shell's `time` command, which on Windows PowerShell corresponds to `Measure-Command { ... }`:

```
$ python3 make_prime1_worst.py
$ time ../cpp/prime1 < prime1_worst.in > out_cpp.txt
$ time python3 ../python/prime1.py < prime1_worst.in > out_py.txt
```

```
wrote prime1_worst.in
real    0m0.020s
real    0m0.188s
```

The C++ program needs 0.02 seconds and the Python program 0.19 seconds for the worst case, and their outputs are identical: 48,320 primes, 4,832 in each range, from 999 900 017 to 999 999 937. The segmented sieve turns a problem that trial division cannot finish in time into one that is done before the interpreter has properly started.

3.7 Signals in the Statement

The size of the input is the strongest signal, but not the only one. The shape of the input and the question asked also point to families of methods, and **Table 3.2** collects the signals that the rest of this book develops. Treat it as a list of first guesses, to be checked against the budget, not as a set of rules.

Table 3.2 Signals in a problem statement and the family of methods they usually point to, with the chapter of this book that treats each family.

Signal in the statement	Family to try first	Chapter
"number of ways", "minimum cost", overlapping choices	dynamic programming	9
a local choice that never needs to be undone; "schedule", "choose the most"	greedy, with an exchange argument	10
"the largest x such that ...", a yes/no test that is monotone in x	binary search on the answer	11
cities and roads, dependencies, grids with walls	graphs: BFS, DFS, shortest paths	12
primes, divisors, "modulo $10^9 + 7$", huge exponents	number theory	13
points, segments, areas	geometry	13
a text and patterns, palindromes	string algorithms	13
many queries on a changing array	data structures: prefix sums, segment trees	6, 11

In this chapter we met the number-theory row twice: FCTRL needed a counting formula for prime factors, and LASTDIG a huge exponent modulo a small number. PRIME1 combined the number-theory signal (primes) with the size signal (a short range of large numbers), and the combination pointed to the segmented sieve.

3.8 Using It for Real

Estimate before you code. For every problem, write in your notebook, next to the restatement of Chapter 2, the largest input, the number of test cases, the time limit, the budget by Equation (3.1), and the step count of your planned method by Equation (3.2). If the method does not fit, do not start coding it; go back to **Table 3.1** and ask what the constraints are hinting at. Five minutes of arithmetic saves an hour of optimizing a method that could never pass.

Do the arithmetic in powers of ten. Estimates need only one significant figure, and three facts make them quick: $2^{10} \approx 10^3$, so $2^{20} \approx 10^6$ and $2^{30} \approx 10^9$; therefore $\log_2 10^3 \approx 10$, $\log_2 10^6 \approx 20$ and $\log_2 10^9 \approx 30$. And $\sqrt{10^{2k}} = 10^k$, so $\sqrt{10^9}$ lies between 10^4 and 10^5, about 3×10^4. With these, $n \log_2 n$ for $n = 10^6$ is about 2×10^7 and for $n = 10^5$ about 1.7×10^6, computed in your head in a few seconds. Write the estimate as a product of powers of ten, add the exponents, and compare with 8, the exponent of the budget: an exponent of 10 means a factor of a hundred too slow, whatever the constant.

Always write the maximum test. A generator for the largest legal input, like `make_fctrl_tests.py` or `make_prime1_worst.py`, takes a few lines and answers the question "is it fast enough?" with a measurement instead of a hope. Measure on your machine, and leave a safety factor of two to five for the judge.

Keep the slow program. A correct but slow program is not wasted: it is the brute force for your stress test. FCTRL's naive loop, trial division for PRIME1 and Python's `pow` for LASTDIG all served that purpose in this chapter.

Outside the judge. The same arithmetic applies to every program that meets real data. Before running a nested loop over two tables of customers, multiply their sizes; before calling a web service once per record, multiply the number of records by the response time. A reporting job at a cooperative that compares every member with every other member is quadratic: fine for 300 members, hopeless for 300,000. The fix is the same as in this chapter: sort, index, or find a formula.

3.9 Common Mistakes, and How to Avoid Them

- **Forgetting the number of test cases.** A method that costs 10^4 steps per case is fine for one case and too slow for 10^5 cases. Multiply, with Equation (3.2).
- **Trusting your laptop's time.** A judge machine may be two or three times slower. Leave a safety factor.
- **Optimizing the wrong thing.** A faster loop does not rescue an algorithm that is a million times over budget. Change the algorithm first; tune later, if at all.
- **Overflow in intermediate values.** `p * p`, `power *= 5`, `result * base`: compute in `long long`, or reduce modulo before multiplying.
- **Printing 1 as a prime, or forgetting 2.** Test ranges that start at 1, 2 and 3.
- **An empty line after every case.** PRIME1 asks for one *between* cases. Print it before every case except the first.
- **Python loops where C loops exist.** Crossing out with a Python `for` loop is slow; slice assignment, `sum`, `sort`, `pow` and `str.join` run in C.
- **A brute force that fails on its own.** The FCTRL brute force hit Python's 4300-digit limit. Run the stress test and read its first failure carefully before blaming the fast program.

3.10 Chapter Summary

- The budget of a problem is its time limit times the steps per second (Equation (3.1)): measured here at about 5×10^8 simple loop rounds per second in C++ and 4×10^6 in Python (**Listing 3.1**, **Listing 3.2**); plan with 10^8 and 10^6 to 10^7.
- The step count of an algorithm follows from three rules: loops multiply, halving gives a logarithm, test cases add up, and all pairs cost $n(n-1)/2$ (Equation (3.3)); the result is compared with the budget at the largest input (Equation (3.2)).
- **Figure 3.1** and **Table 3.1** turn the largest n into the slowest complexity that fits; read backwards, the constraints hint at the intended method.
- FCTRL counts trailing zeros of $n!$ by Legendre's formula (Equation (3.4)) in about 13 divisions per case; the naive loop is correct but was stopped by our judge at the time limit.
- LASTDIG computes $a^b \bmod 10$ by repeated squaring (Equation (3.5)) in at most about 62 multiplications.
- PRIME1 uses a segmented sieve: base primes up to $\sqrt{n}$ (Equation (3.6)), each crossing out its multiples in the range from the start of Equation (3.7); the worst case takes 0.02 s in C++ and 0.19 s in Python.
- Signals in a statement point to families of methods (**Table 3.2**); they are first guesses, checked against the budget.

3.11 Exercises

1. A problem has $T \le 10$ test cases, each an array of $n \le 2 \times 10^5$ numbers, and a time limit of 1 second. For each of $O(n^2)$, $O(n \log n)$ and $O(n)$, compute the total steps with Equation (3.2) and decide whether it fits.
2. Compute $Z(1000)$ with Equation (3.4) by hand, and check the result with `fctrl.cpp`.
3. How many multiplications does `powerMod` perform for $b = 2\,147\,483\,000$? Count the bits of b and the ones among them.
4. Show that the last digit of a^b for $b \ge 1$ depends only on the last digit of a and on $b \bmod 4$, and use this to compute the last digit of 7^{2023} by hand.
5. Run the segmented sieve by hand on the range [90,110], listing which base primes cross out which numbers, and give the primes.
6. Modify `prime1.cpp` so that it prints, for each range, only the number of primes in it. What is the answer for the range $[1, 10^5]$? (Its well-known value is 9,592.)

3.12 Solutions to the Exercises

1. With $T = 10$ and $n = 2 \times 10^5$: $O(n^2)$ gives $10 \cdot 4 \times 10^{10} = 4 \times 10^{11}$ steps, four thousand times the budget of 10^8: no. $O(n \log n)$ gives $10 \cdot 2 \times 10^5 \cdot 18 \approx 3.6 \times 10^7$: yes, with a safety factor of about three. $O(n)$ gives 2×10^6: easily.

2. $\lfloor 1000/5 \rfloor = 200$, $\lfloor 1000/25 \rfloor = 40$, $\lfloor 1000/125 \rfloor = 8$, $\lfloor 1000/625 \rfloor = 1$, and $\lfloor 1000/3125 \rfloor = 0$. So $Z(1000) = 200 + 40 + 8 + 1 = 249$. The program prints 249 for the input `1` followed by `1000`.

3. $2^{30} = 1\,073\,741\,824 \le b < 2^{31}$, so b has 31 bits and the loop runs 31 times, with one squaring each time: 31 squarings. The number of multiplications into the result equals the number of 1 bits of b. Write $b = (2^{31} - 1) - 647$. The number $2^{31} - 1$ is 31 ones in binary, and subtracting $647 = 1010000111_2$ from it simply turns the five 1 bits of 647 into zeros, because no borrowing

happens when every bit is 1. So b has $31 - 5 = 26$ bits equal to 1, and there are 26 multiplications into the result: 57 multiplications in all, and at most 62 for any exponent in range.

4. The last digit of a product depends only on the last digits of the factors, so replacing a by $a \bmod 10$ does not change the answer. For each last digit d, the sequence $d^1, d^2, d^3, \ldots \bmod 10$ repeats with period 1, 2 or 4: for 7 it is 7,9,3,1,7,9,3,1, …; for 2 it is 2,4,8,6, …; for 0, 1, 5, 6 it is constant; for 4 and 9 it has period 2; for 3 and 8 period 4. In every case the period divides 4, so for $b \geq 1$ the answer is the r-th element of the cycle, where $r = ((b - 1) \bmod 4) + 1$. For 7^{2023}: $2022 \bmod 4 = 2$, so $r = 3$ and the answer is the third element of 7,9,3,1: the digit 3. (The case $b = 0$ is special: the answer is 1.)

5. $\sqrt{110} < 11$, so the base primes are 2, 3, 5 and 7. Prime 2 starts at 90 and crosses out the even numbers 90 to 110. Prime 3 starts at $\lceil 90/3 \rceil \cdot 3 = 90$: 90, 93, 96, 99, 102, 105, 108. Prime 5 starts at 90: 90, 95, 100, 105, 110. Prime 7 starts at $\max(49, \lceil 90/7 \rceil \cdot 7) = 91$: 91, 98, 105. The numbers never crossed out are 97, 101, 103, 107 and 109, the primes between 90 and 110.

6. In `primesBetween`, replace the appending of each prime by a counter, return the count, and print it per case. For $[1, 10^5]$ the program prints 9592, which agrees with the well-known value of $\pi(10^5)$, the number of primes up to 10^5.

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] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Singapore: Lulu, 2020.

[3] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

[4] SPOJ, "FCTRL — Factorial." [Online]. Available: https://www.spoj.com/problems/FCTRL/ (accessed Sep. 2026).

[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, Sec. 4.4, "Factorial factors."

[6] SPOJ, "LASTDIG — The last digit." [Online]. Available: https://www.spoj.com/problems/LASTDIG/ (accessed Sep. 2026).

[7] SPOJ, "PRIME1 — Prime Generator." [Online]. Available: https://www.spoj.com/problems/PRIME1/ (accessed Sep. 2026).

Chapter 4

Fast Input and Output

A program with the right algorithm can still exceed the time limit if it spends its seconds reading the input and writing the output. With ten million numbers to read, the difference between the slowest and the fastest way of reading them is a factor of ten in C++ and more in Python, and on a judge that difference is the difference between TLE and AC. This chapter explains where input and output spend their time and measures every common method on the same five million numbers. By the end of it you will be able to explain what a buffer, a system call, synchronization with C's standard I/O and a tied stream are, and why each costs time; choose the right way to read and to write in C++ and in Python for a given amount of data; write a small `fread`-based reader when nothing else is fast enough; avoid the two classic traps, mixing `cin` with `scanf` after unsynchronizing and flushing with `endl` on every line; and solve SPOJ's two I/O tests, INTEST and INOUTEST, in both languages.

4.1 One Trip per Parcel

Picture a courier who must deliver two hundred parcels from a warehouse to one office building across town. A courier who takes one parcel, rides across town, delivers it, and rides back for the next, spends the whole week on the road, although each delivery takes a minute. A courier who loads a van with all two hundred parcels makes one trip. The work at the building is the same; what differs is the number of trips, and the trips are where the time goes.

Input and output work the same way. Reading a number from the input or writing one to the output is cheap; *fetching* input from the operating system or *handing* output to it is expensive, because every such request, a *system call*, leaves the program, enters the operating system kernel, and returns. Fast I/O is about making few trips with a full van: fetching and handing over large blocks at once, and doing the cheap per-number work inside the program.

Rina's team learned this from INTEST, a SPOJ problem whose only purpose is to test reading speed. Their program was a single loop with a division, as simple and fast an algorithm as exists. It read the numbers with plain `std::cin`, and it received *time limit exceeded*. Two lines at the start of `main`, which this chapter explains, made it pass with time to spare.

Who needs this outside the judge? Anybody whose program handles large text files: a log of every transaction of a cooperative for a year, a CSV export from a point-of-sale system, the output of a sensor that writes a line every second. The same program can take a minute or ten minutes on such a file depending on nothing but how it reads and writes, and the rules of this chapter apply unchanged.

4.2 Where the Time Goes

Between your `std::cin >> x` and the bytes in the input file there are three layers, and each can cost time.

The buffer. A program does not ask the operating system for one character at a time. The library keeps a *buffer*, a block of memory of a few kilobytes; when the buffer is empty it asks the

operating system for a whole block with one system call, and then serves characters from memory until the block is used up. Output works in reverse: characters are collected in a buffer and handed to the operating system in one system call when the buffer is full, when the program ends, or when the program *flushes* it on purpose. With buffering, a million numbers need a few hundred system calls instead of millions. The time of an I/O-heavy program is therefore roughly

$$t_{\text{I/O}} \approx c_{\text{byte}} \cdot \text{bytes} + c_{\text{call}} \cdot \text{system calls} \tag{4.1}$$

where c_{call}, the cost of one system call, is hundreds or thousands of times larger than c_{byte}, the cost of handling one byte in memory. Equation (4.1) explains almost every measurement in this chapter: the slow methods make many system calls, or do a lot of work per byte, or both.

Synchronization with C's standard I/O. C++ inherited from C a second family of I/O functions, `scanf` and `printf`, with their own buffers. So that a program may mix the two families freely, reading one number with `scanf` and the next with `std::cin`, the C++ standard requires by default that the C++ streams be *synchronized* with the C streams [1]. In practice, GCC's library implements this by giving `std::cin` and `std::cout` no buffer of their own and passing every single character through the C library, which costs a function call and a lock per character. The call `std::ios::sync_with_stdio(false)` gives up the right to mix the families and, in return, lets the C++ streams use their own fast buffers.

Tied streams. By default `std::cin` is *tied* to `std::cout`: before every input operation, `std::cout` is flushed [2]. The reason is interaction: a program that prints `Your name?` and then reads must make sure the question is on the screen before it waits. On a judge nobody is watching, and with a tie every read of a number flushes the output, possibly a system call each time. `std::cin.tie(nullptr)` cuts the tie.

The courier of Section 4.1 explains all three costs. Synchronization is a rule that every parcel must first be checked in at the central post office, even when all parcels go to the same building. The tie is a rule that the courier must return to the warehouse and report after every single delivery, in case someone there is waiting for news. Buffering is the van. Unsynchronizing and untying abolish the two rules for a courier who works alone, and the van then does its job. None of the rules is foolish: each protects a situation that really occurs, a program that mixes C and C++ I/O, or a person waiting at a keyboard for a prompt. A judge is simply not such a situation, and a program written for the judge may give the protections up.

Flushing on purpose. `std::endl` writes a line break *and flushes* the output buffer [3]. Printing a million lines with `std::endl` makes a million system calls, the courier's million trips. `'\n'` writes the line break only.

Python has the same layers under different names. `input()` reads one line through `sys.stdin`, a text stream, and does extra work per call: it checks whether the input is an interactive terminal, strips the line break, and handles the `input()` machinery. `sys.stdin.readline` skips most of that. `sys.stdin.buffer` is the byte stream underneath the text stream, and `sys.stdin.buffer.read()` fetches the entire input in one call as bytes, without decoding it to text [4]. On output, `print()` converts its arguments, adds separators and a line break, and writes, which is a good deal of work per line; collecting all lines into one string with `str.join` and writing it once does that work in C.

A rough calculation shows the size of the effect. INTEST's largest input is about 110 MB. Read through a 64 KiB buffer, that is $110 \times 10^6 / 65\,536 \approx 1700$ system calls, which at a few microseconds each cost well under a hundredth of a second: with a buffer, the second term of

Equation (4.1) vanishes and only the per-byte work remains. Now write ten million lines, each flushed with `std::endl`: ten million system calls, and at even one microsecond each that is ten seconds of nothing but trips to the operating system. The same program, the same algorithm, the same output, and a factor of a thousand in the number of trips.

4.2.1 What one `std::cin >> x` really does

It helps to see how much happens behind the innocent `std::cin >> x` for an integer `x`. First the stream constructs a small guard object, the *sentry*, which flushes the tied stream if there is one and skips leading whitespace. Then it hands the actual conversion to a *facet* of the stream's *locale*, the object that knows the conventions of a language and region, such as which character is the decimal point and whether digits are grouped with commas. The facet reads the characters one at a time through the stream buffer, collects them, checks them against the number format, converts them, checks for overflow, and sets the error flags of the stream if anything is wrong. Finally the value is stored in `x`. Every one of these steps is reasonable and general, and every one costs a little; with synchronization on, each single character additionally goes through the C library. For five million numbers of ten digits that is fifty million characters, each paying all of it.

Our own reader in Section 4.3 does the minimum instead: it takes the next byte from an array, compares it with `'0'` and `'9'`, and multiplies by ten. It knows nothing about locales, overflow or error flags, which is acceptable only because a judge's input is guaranteed to be well formed. That trade, generality for speed, is the whole secret of hand-made readers.

4.3 Measuring the Ways of Reading

Claims about speed must be measured. We use an input with the shape of INTEST (Section 4.6): a first line with n and k, then n positive integers up to 10^9, one per line. The task is to count how many of the numbers are divisible by k, which is almost no work, so the time is nearly all reading. **Listing 4.1** writes such an input.

Listing 4.1 `gen_intest.py`: writes an INTEST-shaped input with n random numbers.

```
"""gen_intest.py N [SEED]: an INTEST-shaped input with N numbers to stdout."""
import random
import sys


def main():
    n = int(sys.argv[1])
    rng = random.Random(int(sys.argv[2]) if len(sys.argv) > 2 else 1)
    k = rng.randint(1, 50)
    lines = [f"{n} {k}"]
    lines += [str(rng.randint(1, 10**9)) for _ in range(n)]
    sys.stdout.write("\n".join(lines) + "\n")


if __name__ == "__main__":
    main()
```

Listing 4.2 reads that input in four ways, chosen by the number on the command line.

Listing 4.2 `read_bench.cpp`: four ways to read the same input in C++, from plain `cin` to a reader of our own on top of `fread`.

```cpp
// read_bench.cpp METHOD: reads an INTEST-shaped input in one of four ways.
// 1 = cin as it comes, 2 = cin untied and unsynchronized, 3 = scanf,
// 4 = our own reader on top of fread. It prints how many t_i divide by k.
#include <cstdio>
#include <cstdlib>
#include <iostream>

static char buf[1 << 16];
static size_t bufLen = 0, bufPos = 0;

int readByte() {
    if (bufPos == bufLen) {
        bufLen = std::fread(buf, 1, sizeof buf, stdin);  // refill 64 KiB
        bufPos = 0;
        if (bufLen == 0) return EOF;
    }
    return buf[bufPos++];
}

long long readInt() {  // skips spaces, reads an optional '-' and digits
    int c = readByte();
    while (c == ' ' || c == '\n' || c == '\r' || c == '\t') c = readByte();
    bool negative = (c == '-');
    if (negative) c = readByte();
    long long x = 0;
    while (c >= '0' && c <= '9') {
        x = x * 10 + (c - '0');
        c = readByte();
    }
    return negative ? -x : x;
}

int main(int argc, char* argv[]) {
    int method = argc > 1 ? std::atoi(argv[1]) : 2;
    long long n = 0, k = 1, t = 0, count = 0;
    if (method == 1 || method == 2) {
        if (method == 2) {
            std::ios::sync_with_stdio(false);
            std::cin.tie(nullptr);
        }
        std::cin >> n >> k;
        for (long long i = 0; i < n; i++) {
            std::cin >> t;
            if (t % k == 0) count++;
        }
    } else if (method == 3) {
```

Listing 4.2 (Continued) *read_bench.cpp: four ways to read the same input in C++, from plain cin to a reader of our own on top of fread.*

```cpp
        if (std::scanf("%lld %lld", &n, &k) != 2) return 1;
        for (long long i = 0; i < n; i++) {
            if (std::scanf("%lld", &t) != 1) return 1;
            if (t % k == 0) count++;
        }
    } else {
        n = readInt();
        k = readInt();
        for (long long i = 0; i < n; i++) {
            if (readInt() % k == 0) count++;
        }
    }
    std::printf("%lld\n", count);
    return 0;
}
```

Methods 1 and 2 are the same code except for the two lines that unsynchronize and untie. Method 3 uses `scanf`, which parses the format string `"%lld"` on every call; its return value, the number of items read, is checked, which also silences a compiler warning. Method 4 is a reader of our own. `readByte` serves characters from a 64 KiB array `buf` and refills it with one call of `fread` when it is empty, exactly the buffer of Section 4.2 written by hand; `readInt` skips whitespace, reads an optional minus sign, and builds the number digit by digit with $x \leftarrow 10x + d$, the same step as in ADDREV in Chapter 2. It does nothing else: no locale, no format string, no error handling beyond what the judge's well-formed input needs.

To see `readInt` at work, trace it on the characters `-307` followed by a line break, with `buf` holding them:

Table 4.1 The reader `readInt` of **Listing 4.2** on the input `-307`, one character per row.

Character read	Action	`negative`	`x`
space	skipped as whitespace		
`-`	sets the sign, reads the next character	true	0
`3`	$x \leftarrow 10 \cdot 0 + 3$	true	3
`0`	$x \leftarrow 10 \cdot 3 + 0$	true	30
`7`	$x \leftarrow 10 \cdot 30 + 7$	true	307
line break	not a digit: the loop ends	true	307

The function returns -307. The line break that ended the number has been consumed, which is harmless, because the next call skips whitespace anyway. Three comparisons and one multiply-add per digit: that is all.

Listing 4.3 does the same in Python with three methods.

Listing 4.3 `read_bench.py`: three ways to read the same input in Python.

```
"""read_bench.py METHOD: reads an INTEST-shaped input in one of three ways.

1 = input() per line, 2 = sys.stdin.readline per line,
3 = sys.stdin.buffer.read().split() once. Prints how many t_i divide by k.
"""
import sys


def main():
    method = int(sys.argv[1]) if len(sys.argv) > 1 else 3
    count = 0
    if method == 1:
        n, k = map(int, input().split())
        for _ in range(n):
            if int(input()) % k == 0:
                count += 1
    elif method == 2:
        readline = sys.stdin.readline  # look the name up once, not n times
        n, k = map(int, readline().split())
        for _ in range(n):
            if int(readline()) % k == 0:
                count += 1
    else:
        data = sys.stdin.buffer.read().split()  # every token, as bytes
        n, k = int(data[0]), int(data[1])
        for token in data[2:2 + n]:
            if int(token) % k == 0:
                count += 1
    print(count)


if __name__ == "__main__":
    main()
```

Method 2 assigns `sys.stdin.readline` to a local name once; looking up `sys`, then `stdin`, then `readline` five million times is itself measurable in Python. Method 3 reads everything at once and splits it into tokens; `int()` accepts the byte strings that `split()` produces, so no decoding to text is needed.

The script `bench_io.py` in the `tests` folder of this chapter generates five million numbers, half of INTEST's maximum, runs every method three times with the input taken from the file and the output sent to a file, as on a judge, and keeps the best of the three times. It also checks that all readers print the same count. On the machine used for this book, a shared server that was also running other work, which makes all numbers somewhat slower and noisier than on an idle laptop:

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o ../cpp/read_bench ../cpp/read_bench.cpp
$ g++ -std=c++17 -O2 -Wall -Wextra -o ../cpp/write_bench ../cpp/write_bench.cpp
$ python3 bench_io.py
```

```
5,000,000 numbers; best of 3 runs
  read   C++ cin, default                 4.28 s
  read   C++ cin, unsynced + untied       0.59 s
  read   C++ scanf                        0.80 s
  read   C++ fread reader                 0.31 s
  read   Python input()                   6.11 s
  read   Python readline                  1.06 s
  read   Python buffer.read().split()     1.26 s
  write  C++ cout << endl                 4.11 s
  write  C++ cout << '\n'                 0.47 s
  write  C++ printf                       0.56 s
  write  C++ fwrite buffer                0.15 s
  write  Python print()                   1.98 s
  write  Python write per line            1.29 s
  write  Python one join                  1.38 s
all readers agree: True
```

4.3.1 The memory cost of reading everything

Speed is not the only cost. Reading the whole input at once holds all of it in memory, and in Python every token becomes a separate `bytes` object with its own header: `sys.getsizeof(b"123456789")` is 42 bytes for nine characters, plus eight bytes for the pointer in the list. We measured the peak memory of the Python readers on the five-million-number input, a file of 49 MB:

```
237925
method 2 max RSS MB 12
237925
method 3 max RSS MB 320
```

Reading line by line with `readline` needed 12 MB, because each line is discarded after use; reading everything and splitting needed 320 MB, more than six times the size of the file. At INTEST's full size that becomes about 650 MB. Many judges allow that much, but some problems have limits of 64 or 256 MB, and there `readline` is the safe choice. In C++, our `fread` reader needs only its 64 KiB buffer, whatever the size of the input. The general rule: reading everything at once is convenient and fast, but check the memory limit when the input is larger than about 10 MB.

Figure 4.1 draws the same numbers. The slowest method in each panel is red.

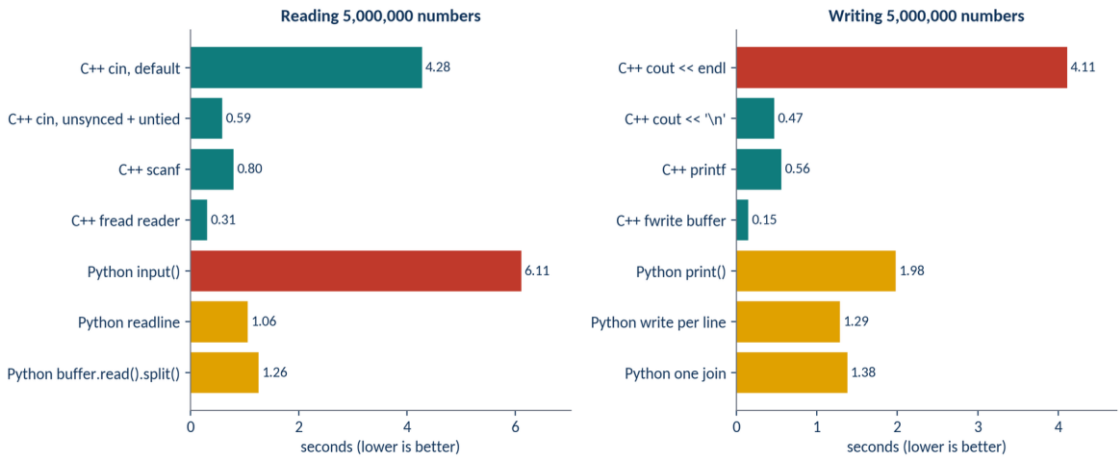


Figure 4.1 Measured time to read and to write five million numbers, best of three runs, C++ in teal and Python in amber, the slowest method in red. The absolute times depend on the machine; the ratios are what matter.

Read the reading panel with Equation (4.1) in mind.

- **Default `cin`: 4.28 s.** Every character goes through the synchronized path, a function call per character. For INTEST's full 10^7 numbers that is over 8 seconds on this machine: TLE on any judge.
- **Unsynchronized and untied `cin`: 0.59 s,** seven times faster, from two lines of code. This is the method to use by default.
- **`scanf`: 0.80 s.** Buffered from the start, but it interprets the format string on every call.
- **Our `fread` reader: 0.31 s,** another factor of two, because it does the least work per byte.
- **Python `input()`: 6.11 s.** Slower than the slowest C++ method, and far too slow for 10^7 lines.
- **Python `readline`: 1.06 s,** and **`buffer.read().split()`: 1.26 s.** Both are about six times faster than `input()`. On this input, with one number per line, reading line by line happens to be a little faster than splitting everything at once, because `split()` builds a list of five million byte strings in memory before the loop starts; on inputs with many numbers per line the reading-all method is the more convenient and usually the faster one.

Why is `scanf` slower than unsynchronized `cin` here, although both are buffered? Both make few system calls, so the second term of Equation (4.1) is small for both; the difference lies in the first term, the work per byte. `scanf` must read and interpret its format string on every call, and it goes through the general machinery of the C library for every number. Unsynchronized `cin` knows the type of its target at compile time and skips the format parsing. Our `fread` reader does even less per byte, and it wins. On another compiler or library the order of `scanf` and `cin` may be reversed; the lesson that transfers is the method, not the ranking: measure, and explain the measurement with the two terms of Equation (4.1).

> **WARNING.** After `std::ios::sync_with_stdio(false)`, never mix `std::cin`/`std::cout` with `scanf`/`printf` (or `getchar`, `puts`) in the same program. The two families now have separate buffers, and a number that one family has already pulled into its buffer is invisible to the other: input is skipped or read twice, output appears out of order, and no compiler warns you. Choose one family per program.

4.4 Measuring the Ways of Writing

Listing 4.4 writes the numbers 1 to N, one per line, in four ways.

Listing 4.4 `write_bench.cpp`: four ways to write the same output in C++.

```cpp
// write_bench.cpp METHOD N: prints the numbers 1..N, one per line, in one of
// four ways. 1 = cout with endl, 2 = cout with '\n' (unsynchronized),
// 3 = printf, 4 = our own buffer written with fwrite.
#include <cstdio>
#include <cstdlib>
#include <iostream>

static char out[1 << 16];
static size_t outLen = 0;

void flushOut() {
    std::fwrite(out, 1, outLen, stdout);
    outLen = 0;
}

void writeInt(long long x) {  // x >= 0 here; digits are made right to left
    char digits[24];
    int len = 0;
    do {
        digits[len++] = static_cast<char>('0' + x % 10);
        x /= 10;
    } while (x > 0);
    if (outLen + len + 1 > sizeof out) flushOut();
    while (len > 0) out[outLen++] = digits[--len];
    out[outLen++] = '\n';
}

int main(int argc, char* argv[]) {
    int method = argc > 1 ? std::atoi(argv[1]) : 2;
    long long n = argc > 2 ? std::atoll(argv[2]) : 10;
    if (method == 1) {
        for (long long i = 1; i <= n; i++) std::cout << i << std::endl;
    } else if (method == 2) {
        std::ios::sync_with_stdio(false);
        for (long long i = 1; i <= n; i++) std::cout << i << '\n';
    } else if (method == 3) {
        for (long long i = 1; i <= n; i++) std::printf("%lld\n", i);
    } else {
        for (long long i = 1; i <= n; i++) writeInt(i);
        flushOut();
    }
    return 0;
}
```

`writeInt` is the output counterpart of `readInt`: it produces the digits of x from right to left into a small array with `x % 10` and `x / 10`, copies them in the right order into a 64 KiB output buffer, and hands the buffer to the operating system with one `fwrite` when it is nearly full; `flushOut` at the end writes the rest. It handles only non-negative numbers, which is all this program needs; Exercise 4 adds the minus sign.

Listing 4.5 `write_bench.py`: three ways to write the same output in Python.

```
"""write_bench.py METHOD N: prints 1..N, one per line, in one of three ways.

1 = print() per line, 2 = sys.stdout.write per line, 3 = one join and one write.
"""
import sys


def main():
    method = int(sys.argv[1]) if len(sys.argv) > 1 else 3
    n = int(sys.argv[2]) if len(sys.argv) > 2 else 10
    if method == 1:
        for i in range(1, n + 1):
            print(i)
    elif method == 2:
        write = sys.stdout.write
        for i in range(1, n + 1):
            write(f"{i}\n")
    else:
        sys.stdout.write("\n".join(map(str, range(1, n + 1))) + "\n")


if __name__ == "__main__":
    main()
```

The writing panel of **Figure 4.1** tells the story of `std::endl`: 4.11 s against 0.47 s with `'\n'`, nearly nine times slower, for one character of difference in the source. With output going to a file, every flush is a system call, five million of them. The hand-made buffer is the fastest at 0.15 s. In Python, `print()` costs 1.98 s, and writing each line with `sys.stdout.write` 1.29 s. Joining all lines into one string and writing it once took 1.38 s here, slightly more than writing line by line, because the joined string of about 39 megabytes must be built in memory first; on a less busy machine the two are usually close, and joining wins clearly over `print()` in every case. Chapter 3's programs and the solutions below collect their output and write it once.

4.5 The Rules

The measurements condense into a short list that covers nearly every problem.

C++.

1. Start `main` with `std::ios::sync_with_stdio(false);` and `std::cin.tie(nullptr);` whenever the input or output has more than about 10^5 numbers. It costs nothing when the input is small, so many competitors write it in every program.
2. End lines with `'\n'`, never with `std::endl`.
3. After rule 1, use only `std::cin` and `std::cout`. If you prefer `scanf` and `printf`, skip rule 1 and use only them.
4. For more than about 10^7 numbers, or a time limit that is tight for the algorithm, use an `fread` reader like the one of **Listing 4.2**, and collect the output in a buffer or a `std::string` written once.

Python.

1. Never use `input()` for more than about 10^5 lines.
2. Read with `data = sys.stdin.buffer.read().split()` and walk through `data` with an index, or with `readline = sys.stdin.readline` when line structure matters.
3. Collect the answers in a list and write `"\n".join(answers)` once, or use `sys.stdout.write`; avoid `print()` in loops over large outputs.

4. `print(*answers, sep="\n")` is a convenient one-line alternative to joining; it converts every answer inside a single call, which is far better than `print()` in a loop, though building the joined string yourself gives you more control over the last line break.
5. Keep the per-number work in C: `map(int, ...)`, `sum(...)`, slice assignment, `sort`.

> **KEY INSIGHT.** Fast I/O is not a replacement for the right algorithm. Chapter 3's budget comes first: an $O(n^2)$ method does not pass because it reads quickly. Fast I/O matters when the algorithm is already right and the input or output is large, and then it can matter by a factor of ten. The famous warning that premature optimization is the root of much evil [5] applies to hand-made readers too: use the two lines of rule 1 always, and the `fread` reader only when the measurement says so.

Mixing tokens and lines. Some inputs mix numbers and whole lines of text, for example a count followed by names that contain spaces. In C++, `std::cin >> n` reads the number and leaves the line break after it in the buffer; a following `std::getline(std::cin, line)` then reads the *rest of that line*, which is empty, instead of the next line. The cure is to discard the rest of the line after reading the number, with `std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n')` from the header `<limits>`, or simply to read every line with `std::getline` and convert the numbers yourself with `std::stoi`. In Python the problem does not arise with `readline`, which always returns whole lines; with the read-everything method, use `sys.stdin.buffer.read().split(b"\n")` to split into lines rather than tokens when spaces inside a line matter.

Interactive problems are the exception. A few problems are *interactive*: the program asks a question, the judge's program answers, and the next question depends on the answer. Here a buffered question that has not been flushed never reaches the judge, and both programs wait for each other forever, which the judge reports as TLE. In such problems, flush after every question: `std::cout << question << std::endl;` in C++ (the one legitimate use of `endl`), or `print(question, flush=True)` in Python.

4.6 INTEST: Enormous Input

The problem (SPOJ INTEST, "Enormous Input Test", `https://www.spoj.com/problems/INTEST/`) [6]. The first line contains two positive integers n and k, both at most 10^7. Then follow n lines, each with a positive integer $t_i \le 10^9$. Print how many of the t_i are divisible by k. The statement says outright that the problem tests the speed of input, and that the program should be able to process at least 2.5 MB of input per second at runtime.

Reading the constraints. n up to 10^7 lines of up to 10 digits: up to about 110 MB of input. The algorithm is one pass with one division per number, $O(n)$, 10^7 steps: nothing. Everything is in the reading. By the measurements of Section 4.3, reading 10^7 numbers takes over 8 seconds with default `cin` on our machine, about 1.2 seconds with unsynchronized `cin`, and about 0.6 seconds with the `fread` reader. The C++ solution uses the `fread` reader.

Listing 4.6 `intest.cpp`: INTEST with an `fread`-based reader of non-negative numbers.

```cpp
// SPOJ INTEST (Enormous Input Test): count the t_i divisible by k, read fast.
#include <cstdio>

static char buf[1 << 16];
static size_t bufLen = 0, bufPos = 0;

int readByte() {
    if (bufPos == bufLen) {
        bufLen = std::fread(buf, 1, sizeof buf, stdin);
        bufPos = 0;
        if (bufLen == 0) return EOF;
    }
    return buf[bufPos++];
}

long long readUnsigned() {  // INTEST has no negative numbers
    int c = readByte();
    while (c != EOF && (c < '0' || c > '9')) c = readByte();
    long long x = 0;
    while (c >= '0' && c <= '9') {
        x = x * 10 + (c - '0');
        c = readByte();
    }
    return x;
}

int main() {
    long long n = readUnsigned(), k = readUnsigned();
    long long count = 0;
    for (long long i = 0; i < n; i++) {
        if (readUnsigned() % k == 0) count++;
    }
    std::printf("%lld\n", count);
    return 0;
}
```

This reader is a little simpler than the one of **Listing 4.2**: INTEST has no negative numbers, so `readUnsigned` skips every character that is not a digit, which also makes it immune to `\r` characters from Windows line endings and to any other separator. It stops at end of file (`EOF`), so a missing final line break does no harm.

Listing 4.7 `intest.py`: INTEST in Python, reading all tokens at once.

```python
"""SPOJ INTEST (Enormous Input Test): count the t_i divisible by k, read fast."""
import sys


def main():
    data = sys.stdin.buffer.read().split()
    n, k = int(data[0]), int(data[1])
    print(sum(1 for token in data[2:2 + n] if int(token) % k == 0))


if __name__ == "__main__":
    main()
```

The Python version reads all tokens at once and counts with a generator expression inside `sum`. On the sample, seven numbers with $k = 3$ of which four are divisible:

```
$ printf '7 3\n1\n51\n966369\n7\n9\n999996\n11\n' | ./cpp/intest
4
```

The Python program also prints `4`. On the five-million-number input of Section 4.3, whose answer the plain `brute_intest.py` (reading with `input()`) also computes:

```
$ cd tests
$ time ../cpp/intest < big_intest.in
$ time python3 ../python/intest.py < big_intest.in
$ python3 brute_intest.py < big_intest.in
```

```
237925
real    0m0.324s
237925
real    0m1.496s
237925
```

All three agree on the count. The C++ program needs 0.32 seconds for five million numbers, so about 0.65 seconds for INTEST's maximum of ten million, well inside the budget. The Python program needs 1.5 seconds, so about 3 seconds for the maximum, which may or may not fit, depending on the judge's time limit and the speed of its older Python; this is one of the problems where Python is on the edge, and where C++ with fast I/O is the safe choice.

Put the statement's figure of 2.5 MB per second next to these measurements. The five-million-number file has 49 MB; the C++ program read it in 0.32 seconds, about 150 MB per second, sixty times the stated requirement, and the Python program in 1.5 seconds, about 33 MB per second. The default `cin` of Section 4.3 managed about 11 MB per second on the same file. The requirement sounds modest, but it was written for judge machines of many years ago, and the time limit of the problem is set so that only the careful methods pass. Converting a time into megabytes per second, as here, is a quick way to compare your measurement with such a statement.

What the judge's clock measures. Our timings are *wall-clock* times: the time from start to end as a clock on the wall would show it, including the time the program spends waiting for the disk or the operating system. Many judges measure *CPU time* instead: only the time the processor actually spends executing the program. For a program that computes, the two are nearly equal; for an I/O-heavy program they can differ, and on a busy judge machine the wall-clock time can be much larger than the CPU time. This is one more reason to leave a safety factor, and it explains why the same submission can show slightly different times on different days.

4.7 INOUTEST: Enormous Input and Output

The problem (SPOJ INOUTEST, "Enormous Input and Output Test", `https://www.spoj.com/problems/INOUTEST/`) [7]. The first line contains N, of the order of a million. Each of the next N lines contains two integers, which may be negative. For each pair print their product on its own line. Now both directions are large: a million lines in, a million lines out.

Reading the constraints. The values are small enough that their products fit easily in 64 bits; we compute in `long long` regardless, because a product of two numbers is the classic place where a 32-bit type overflows. Negative numbers are allowed, so a reader for this problem must handle a minus sign, and so must a writer. The C++ solution uses unsynchronized streams for input, which is fast enough for two million numbers, and builds the whole output in one `std::string`, which is written with a single `<<`.

The size of the output deserves the same estimate as the input. In our generated test the factors reach 65 536 in absolute value, as in the third line of the small test below, so a million products have up to ten digits and a sign, so the output is up to about 12 MB, and with one `std::endl` per line it would cost a million system calls. Collecting it in one string costs 12 MB of memory, well within any limit, and one large write. The same estimate for a problem with ten million output

lines would say 120 MB, and then writing in blocks through a buffer, as `writeInt` does, is better than holding everything at once.

Listing 4.8 `inoutest.cpp`: INOUTEST with unsynchronized streams and the output collected in one string.

```cpp
// SPOJ INOUTEST (Enormous Input and Output Test): print a * b for every pair.
#include <iostream>
#include <string>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n;
    std::cin >> n;
    std::string out;
    out.reserve(static_cast<size_t>(n) * 12);  // room for n short lines
    for (int i = 0; i < n; i++) {
        long long a, b;
        std::cin >> a >> b;
        out += std::to_string(a * b);
        out += '\n';
    }
    std::cout << out;  // one large write instead of n small ones
    return 0;
}
```

The call `out.reserve(...)` allocates room for about twelve characters per line in advance, so that the string does not have to grow and be copied repeatedly as lines are appended; it is an optimization of convenience, not of necessity.

Listing 4.9 `inoutest.py`: INOUTEST in Python, reading all tokens and joining all products.

```python
"""SPOJ INOUTEST (Enormous Input and Output Test): print a * b for every pair."""
import sys


def main():
    data = sys.stdin.buffer.read().split()
    n = int(data[0])
    products = (int(data[1 + 2 * i]) * int(data[2 + 2 * i]) for i in range(n))
    sys.stdout.write("\n".join(map(str, products)) + "\n")


if __name__ == "__main__":
    main()
```

The generator expression `products` computes the products one at a time as `join` consumes them, so the program never holds a second list of a million numbers. A small test with a negative number and a large product:

```
$ printf '3\n2 3\n-4 5\n65536 -65536\n' | ./cpp/inoutest
6
-20
-4294967296
```

The third product, -2^{32}, does not fit in a 32-bit `int`: an `int` product would have printed `0`, the kind of silent overflow that gives WA on large tests only. The Python program prints the same three lines. On a million random pairs made by `gen_inoutest.py`, both programs and the plain `brute_inoutest.py` (with `input()` and `print()`) write identical files:

```
$ python3 gen_inoutest.py 1000000 3 > big_inoutest.in
$ time ../cpp/inoutest < big_inoutest.in > o1.txt
$ time python3 ../python/inoutest.py < big_inoutest.in > o2.txt
$ python3 brute_inoutest.py < big_inoutest.in > o3.txt
$ cmp o1.txt o3.txt && cmp o2.txt o3.txt && echo "all three outputs identical"
```

```
real    0m0.359s
real    0m0.788s
all three outputs identical
```

A million lines in and out in 0.36 seconds in C++ and 0.79 seconds in Python. Here Python passes comfortably: the input is a tenth of INTEST's, and the reading-all and joining methods keep the per-line work in C.

4.8 Portability Notes

A few details differ between systems and judges, and they bite at the worst moment.

- **`getchar_unlocked`.** Many fast-reading templates found online use `getchar_unlocked`, a POSIX function that reads a character without locking. It does not exist in the Windows C library, so such a template does not compile with MSYS2 on Windows. The `fread` reader of this chapter is standard C and works everywhere, at the same speed.
- **Line endings.** Files written on Windows end each line with `\r\n`. Token-based readers (`std::cin >>`, `split()`, and our readers, which skip non-digits) ignore the `\r`. A program that reads whole lines with `std::getline` and compares them with a string sees the `\r` and fails; strip it.
- **PowerShell pipes.** `Get-Content file | program` in PowerShell reads the file as text, line by line, and re-encodes it before passing it on, which is slow for large files and can change the bytes. For timing and for large inputs, use Command Prompt's `program < file`, or our `bench_io.py` and `minijudge.py`, which pass the file to the program directly.
- **Timing with the shell.** `time` on Linux and macOS, `Measure-Command { cmd /c "program < file" }` in PowerShell. Always time with output going to a file, not to the terminal: a terminal draws every character on the screen and is far slower than any method in this chapter.
- **Trailing spaces and the last line break.** Most judges, SPOJ's default checker among them, ignore spaces at the end of a line and a missing line break at the end of the output, but a few special checkers do not. Code that prints `" ".join(values)` or a line break after every line produces clean output and is never wrong; code that prints a space after every number relies on the checker's tolerance.
- **Python's unbuffered mode.** If the environment variable `PYTHONUNBUFFERED` is set, or Python is started with `-u`, output is not buffered and every `print()` becomes a system call. Some development environments set it; our first benchmark run happened in such an environment and measured `print()` at more than eight times its real cost. `bench_io.py` removes the variable for the programs it runs.

4.9 Using It for Real

Two templates. Keep a C++ template that starts with the two lines of rule 1 and a Python template that starts with `data = sys.stdin.buffer.read().split()` and ends with one `sys.stdout.write("\n".join(out) + "\n")`. Starting every program from them makes fast I/O a habit that costs no thought, and it removes the whole class of TLE this chapter is about. Keep the `fread` reader of **Listing 4.6** in the same place, for the problems that need it. The standard training books give very similar templates [8].

A checklist before submitting an I/O-heavy program.

1. Count the input: numbers times digits gives the bytes; above about 1 MB, fast I/O matters, above about 10 MB it decides.
2. C++: the two lines of rule 1 are present, and no `scanf` or `printf` hides anywhere in the program, not even in a debugging line.
3. No `std::endl` in any loop; in Python, no `print()` in a loop over the output.
4. The output is collected and written in one piece, or at least through a buffered stream.
5. For a very large input in Python, the memory of reading everything at once, about six times the file size, fits the memory limit.
6. The maximum test, made by a generator, runs on your machine in well under half the time limit, measured with the output redirected to a file.

Six questions, one minute, and the whole family of I/O verdicts is gone before the judge sees the program.

Measure your own machine. Run `bench_io.py` once on your computer. The absolute numbers tell you how fast your machine is compared with ours; the ratios will be similar, and seeing `std::endl` take nine times as long on your own screen is more convincing than any book.

Large files at work. A cooperative's yearly transaction log, a sensor's readings, an export from a shop's cash register: the programs that process them are I/O programs, and the rules of Section 4.5 apply as they are. Read in large blocks, split once, keep the per-record work in the fast parts of the language, collect the output, and write it once. A script that summarizes a million-line CSV file should take seconds, not minutes; if it takes minutes, look at how it reads and writes before looking at anything else.

4.10 Common Mistakes, and How to Avoid Them

- **`std::endl` in a loop.** It flushes every line; use `'\n'`. Nine times slower in our measurement.
- **Forgetting `sync_with_stdio(false)`.** Seven times slower reading with GCC. Put it in your template.
- **Mixing `cin` and `scanf` after unsynchronizing.** Input is lost or duplicated. One family per program.
- **Unsynchronizing in an interactive problem and never flushing.** The judge never sees the question; flush after each one.
- **`input()` for a million lines in Python.** Six times slower than `readline` here. Read everything at once, or use `readline`.
- **`print()` in a loop for a large output.** Collect and write once.
- **`int` products.** $65536 \cdot 65536$ overflows an `int`. Use `long long` for products.
- **`getline` after `cin >>`.** The leftover line break makes `getline` return an empty line. Discard the rest of the line first.
- **Reading a huge input at once into Python objects.** 320 MB for a 49 MB file in our measurement. Check the memory limit, or read line by line.
- **A reader without a minus sign.** INOUTEST has negative numbers; a reader that ignores `-` turns -4 into 4 and gives WA.
- **Timing output to a terminal.** The terminal is the bottleneck. Redirect output to a file when measuring.

4.11 Chapter Summary

- I/O time is roughly bytes times a small cost plus system calls times a large cost (Equation (4.1)); buffering reduces the number of system calls, and fast methods do little work per byte.
- By default, GCC's `std::cin` and `std::cout` are synchronized with C's I/O and `std::cin` is tied to `std::cout`; `std::ios::sync_with_stdio(false)` and `std::cin.tie(nullptr)` remove both costs, after which the two I/O families must not be mixed.
- Measured on five million numbers (**Figure 4.1**): reading took 4.28 s with default `cin`, 0.59 s unsynchronized, 0.80 s with `scanf` and 0.31 s with an `fread` reader (**Listing 4.2**); Python took 6.11 s with `input()` and about 1.1 to 1.3 s with `readline` or reading everything at once (**Listing 4.3**).
- Writing took 4.11 s with `std::endl` and 0.47 s with `'\n'` (**Listing 4.4**); Python's `print()` took 1.98 s against about 1.3 s for writing strings directly (**Listing 4.5**).
- INTEST is solved with an `fread` reader in C++ (**Listing 4.6**) and with one read and split in Python (**Listing 4.7**), at 0.32 s and 1.5 s for five million numbers; Python is on the edge at the full size.
- INOUTEST collects the output in one string (**Listing 4.8**, **Listing 4.9**) and computes products in 64 bits; a million lines take 0.36 s in C++ and 0.79 s in Python.
- Reading everything at once costs memory: 320 MB for a 49 MB input in Python, against 12 MB line by line; check the memory limit.
- Interactive problems must flush after every question; `getchar_unlocked` is not portable to Windows; timing must redirect output to a file.

4.12 Exercises

1. A problem has 3×10^6 numbers in the input and a time limit of 1 second. Using the measurements of Section 4.3, which C++ reading methods are safe, which are risky, and which will fail? Answer the same question for Python.

2. Explain, using Equation (4.1), why writing five million lines with `std::endl` is slow when the output goes to a file, but hardly slower than `'\n'` when a program writes only ten lines.

3. A program calls `std::ios::sync_with_stdio(false)`, reads n with `std::cin`, and then reads the n numbers with `scanf`. Describe what can go wrong, and give two ways to fix the program.

4. Extend `writeInt` in **Listing 4.4** so that it also writes negative numbers correctly. Be careful with the most negative `long long`, -2^{63}, whose absolute value does not fit in a `long long`.

5. Rewrite `inoutest.py` so that it prints with `print()` in a loop, run it on the million pairs of Section 4.7, and compare the time with that of **Listing 4.9**.

6. In a few sentences, explain why an interactive problem needs a flush after every question, and what verdict you get if you forget it.

7. Using the memory measurement of Section 4.3.1, estimate the peak memory of `intest.py` on INTEST's largest input, and rewrite it with `readline` so that it needs almost no memory. How much slower do you expect it to be?

8. Trace `readInt` of **Listing 4.2**, as in **Table 4.1**, on the input `\n\n42` (two line breaks, the digits 4 and 2, a space). What does it return, and which character is the next one `readByte` will deliver?

4.13 Solutions to the Exercises

1. Scaling the five-million measurements by 3/5: default `cin` about 2.6 s, fail; unsynchronized `cin` about 0.35 s, safe; `scanf` about 0.5 s, safe but with less margin; the `fread` reader about 0.2 s, safe. In Python, `input()` about 3.7 s, fail; `readline` or reading everything at once about 0.65 to 0.75 s, risky on a slower judge; the work on each number then comes on top.

2. With `std::endl`, the number of system calls equals the number of lines. For five million lines, the term $c_{\text{call}} \cdot$ system calls of Equation (4.1) is five million times the cost of a system call, which dominates everything else. With ten lines, it is ten system calls, a few microseconds; the difference from `'\n'` is invisible. `std::endl` is not slow in itself; it is slow when multiplied by millions.

3. After unsynchronizing, `std::cin` has its own buffer. When it reads n, it fetches a whole block of the input into that buffer, which contains not only n but also the first many numbers. `scanf` reads from the C library's buffer, which has not seen those numbers, so it starts reading after the block that `std::cin` took: the first numbers are skipped, and the program reads wrong values or runs out of input. Fix 1: read everything with `std::cin`. Fix 2: remove the `sync_with_stdio(false)` line and read everything with `scanf` (or keep synchronization and mix, accepting the slow `cin`).

4. Handle the sign first and then work with the magnitude as an `unsigned long long`, which can hold 2^{63}:

```
void writeInt(long long x) {
    char digits[24];
    int len = 0;
    unsigned long long u = x < 0 ? 0ULL - static_cast<unsigned long long>(x)
                                 : static_cast<unsigned long long>(x);
    do {
        digits[len++] = static_cast<char>('0' + u % 10);
        u /= 10;
    } while (u > 0);
    if (x < 0) digits[len++] = '-';  // written first, since digits run backwards
    if (outLen + len + 1 > sizeof out) flushOut();
    while (len > 0) out[outLen++] = digits[--len];
    out[outLen++] = '\n';
}
```

The expression `0ULL - static_cast<unsigned long long>(x)` computes the magnitude with unsigned arithmetic, which wraps around modulo 2^{64} and is therefore well defined even for $x = -2^{63}$, whereas `-x` would overflow.

5. Replace the last two lines of `main` by `for i in range(n): print(int(data[1 + 2 * i]) * int(data[2 + 2 * i]))`. On the machine of this book, with Python's normal buffering, two runs of each gave 0.74 s and 0.98 s for the `print()` version against 0.60 s and 0.78 s for **Listing 4.9**: about a quarter slower, and the outputs were identical. Here the difference is modest because reading and multiplying take much of the time; the more lines a program prints, the more `print()` costs. With `PYTHONUNBUFFERED` set, it becomes many times slower, which is worth trying once to see the cost of a system call per line.

6. In an interactive problem, the judge's program reads the question before it can answer, and our program reads the answer before it can ask the next question. If the question stays in our output buffer, the judge never receives it and waits; our program waits for the answer; neither proceeds, until the time limit expires. The verdict is TLE (some judges call it "idleness limit exceeded"), although the program's logic may be perfect. Flushing after each question hands it to the operating system immediately.

7. The five-million-number file of 49 MB needed about 320 MB; INTEST's largest input has twice as many numbers, so about 650 MB. With `readline`, the program keeps one line at a time:

```
import sys

def main():
    readline = sys.stdin.readline
    n, k = map(int, readline().split())
    print(sum(1 for _ in range(n) if int(readline()) % k == 0))

main()
```

Its memory stays at about a dozen megabytes whatever n is. By Section 4.3 its speed is about the same as reading everything at once, 1.06 s against 1.26 s for five million numbers on our machine, so on this input shape nothing is lost.

8. The two line breaks are skipped as whitespace; `4` gives $x = 4$, `2` gives $x = 42$; the space is not a digit and ends the loop. The function returns 42, and the space has been consumed, so the next character `readByte` delivers is whatever follows the space, or `EOF` if nothing does.

References

[1] cppreference.com, "std::ios_base::sync_with_stdio." [Online]. Available: https://en.cppreference.com/w/cpp/io/ios_base/sync_with_stdio (accessed Sep. 2026).

[2] cppreference.com, "std::basic_ios::tie." [Online]. Available: https://en.cppreference.com/w/cpp/io/basic_ios/tie (accessed Sep. 2026).

[3] cppreference.com, "std::endl." [Online]. Available: https://en.cppreference.com/w/cpp/io/manip/endl (accessed Sep. 2026).

[4] Python Software Foundation, "sys — System-specific parameters and functions: sys.stdin, sys.stdout," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/sys.html#sys.stdin (accessed Sep. 2026).

[5] D. E. Knuth, "Structured programming with go to statements," *ACM Computing Surveys*, vol. 6, no. 4, pp. 261–301, 1974.

[6] SPOJ, "INTEST — Enormous Input Test." [Online]. Available: https://www.spoj.com/problems/INTEST/ (accessed Sep. 2026).

[7] SPOJ, "INOUTEST — Enormous Input and Output Test." [Online]. Available: https://www.spoj.com/problems/INOUTEST/ (accessed Sep. 2026).

[8] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Singapore: Lulu, 2020.

Chapter 5

Compiler Flags, Pragmas and Portability

A program is not judged on your computer. It is judged on a machine you never see, built by a compiler you did not choose, with switches you did not set. This chapter is about that gap. By the end of it you will be able to say exactly what happens between the moment you press *Submit* and the moment your program starts reading its input; choose the compiler flags that find bugs at home and the ones that make code fast; tell a safe pragma from a gamble and measure the difference instead of believing a blog; recognize the handful of language features that behave differently on Windows, macOS, Linux and the judge; write a program for the SPOJ problem FCTRL that is correct everywhere; and run a small script that compiles your code under several standards, compilers and checking modes before you submit.

5.1 A Program That Worked at Home

Sinta is in her last year of senior high school in Malang and is preparing for the provincial round of the national science olympiad in informatics. Every evening after dinner she solves two or three problems on SPOJ on the family laptop, a Windows machine that her younger brother also uses for school. Her friend Bima, who trains with her on weekends, has a MacBook. They share solutions through a chat group and compare their speed.

One Saturday they both solve FCTRL, a classic SPOJ problem about the number of zeros at the end of a factorial. Sinta's program is accepted on the first try. She sends it to Bima so that he can compare it with his, and the program does not even compile on his Mac: the very first line, `#include <bits/stdc++.h>`, names a file that his compiler does not have. Bima fixes the includes, and the program prints the right answers. Then he adds a line he found on a competitive-programming blog, `#pragma GCC target("avx512f")`, which is supposed to make loops faster, and submits the program to SPOJ. The verdict is *Runtime error (SIGILL)*. The same source, correct on two laptops, crashes on the judge.

The week after, Sinta writes a program that uses `long` for a value that can reach ten billion. On the judge it is accepted; on her laptop it prints negative numbers. She spends an evening looking for a logic error that does not exist: on Windows a `long` has 32 bits, on the Linux judge it has 64.

None of these three failures has anything to do with algorithms. The algorithm was right each time. What differed was the *environment*: the compiler, its version, the switches it was given, the operating system and the processor. Everyone who writes a program that runs somewhere else meets this gap, not only contestants. It is the same gap a cook meets when a cake recipe that works perfectly in her own oven burns in her neighbor's, because the neighbor's oven runs twenty degrees hotter and the recipe never said so. The fix in both cases is the same: know which parts of your result depend on your own kitchen, write them down, and test in more than one kitchen before you rely on the recipe.

This chapter gives you that knowledge for C++ and Python on SPOJ. It is short on theory and long on experiments, because every claim about compilers can be checked in a few seconds, and

should be. All the experiments were run in September 2026 on a Linux virtual machine with two cores of a 2.1 GHz Intel Xeon processor, with g++ 13.3, clang 18.1 and CPython 3.11; the programs are written so that they behave the same with the newer versions installed in Chapter 1.

5.2 The Idea in Plain Words

5.2.1 From source text to a running program

A C++ source file is text. A processor cannot run text; it runs *machine code*, a sequence of numbered instructions such as "add these two registers" or "jump back eight bytes if the last comparison said less". The **compiler** translates the text into machine code once, before the program runs. Python works differently: the text is translated into an intermediate form each time the program starts, and an **interpreter** carries it out step by step. On a judge, both steps happen on the judge's machine.

Table 5.1 lists what happens after you submit a C++ program to SPOJ, and which of the steps you control.

Table 5.1 What happens to a C++ submission on SPOJ, and who controls each step.

Step	What happens	Who decides
1. Language choice	You pick an entry such as "C++14 (gcc 8.3)" from a menu	You
2. Compilation	The judge runs its compiler on your text with its own flags	The judge
3. Execution	The machine code runs on the judge's processor, once per test file	The judge's hardware
4. Limits	Time and memory are measured; the program is stopped if it exceeds them	The problem setter
5. Comparison	Your output is compared with the expected output	The judge's checker

Only the first step and the text itself are yours. Everything this chapter discusses is about writing text that survives steps 2 and 3 on a machine that is not yours.

5.2.2 Flags, pragmas, portability

> **DEFINITION.** A *compiler flag* is a switch given to the compiler on its command line, such as `-O2` (optimize) or `-std=c++17` (use the 2017 C++ standard). A *pragma* is a line inside the source file, starting with `#pragma`, that asks the compiler to change its behavior from that point on. A program is *portable* when it compiles and gives the same output with every compiler, operating system and processor it is meant for. *Undefined behavior* is what the C++ standard calls an operation, such as a signed integer overflow or reading past the end of an array, for which the language promises nothing at all; the compiler may assume it never happens.

The difference between a flag and a pragma is who writes it. You cannot change the flags the judge uses, but you can write pragmas, because they are part of your text. That is why pragmas are popular in competitive programming, and why they are dangerous: a pragma that tells the compiler "this processor has instructions X and Y" is a promise about a machine you have never seen.

5.2.3 Three kinds of difference

When a program behaves differently at home and on the judge, the cause is almost always one of three things.

1. **The language is different.** Your compiler understands C++20, the judge's C++14; your standard library has a header that the judge's lacks; your compiler accepts an extension that another rejects.
2. **The sizes are different.** A `long` has 32 bits on Windows and 64 bits on Linux and macOS; a `char` is signed on one processor and unsigned on another; the default stack is 1 MB on Windows and 8 MB on most Linux systems.
3. **The program has undefined behavior.** A signed overflow, an uninitialized variable or an index out of range can give the right answer with one compiler and one set of flags and a wrong answer, or no answer, with another. This is the most treacherous kind, because the program *looks* right on the machine where you test it.

The rest of the chapter takes the three kinds one at a time, on one real problem.

5.3 The Problem We Carry Through: FCTRL

5.3.1 The statement in our own words

SPOJ problem FCTRL, "Factorial" [1], asks a question about the number $N! = 1 \times 2 \times \cdots \times N$. The input starts with a number T of test values, about 100,000 of them, and then gives T positive integers N, each at most 1,000,000,000. For each N the program must print $Z(N)$, the number of zeros at the end of the decimal form of $N!$. For instance $5! = 120$ ends in one zero and $10! = 3628800$ in two. The time limit on the problem page is 6 seconds for the whole file, and the memory limit is 1536 MB, the default on SPOJ.

Computing $N!$ itself is out of the question: 1000! already has 2568 digits, and N goes up to a billion. We need a way to count the zeros without building the number.

5.3.2 The idea: count the fives

A zero at the end of a number is a factor 10, and $10 = 2 \times 5$. Every zero at the end of $N!$ therefore needs one factor 2 and one factor 5 from the product $1 \times 2 \times \cdots \times N$. Factors 2 are far more plentiful than factors 5 (every second number is even, only every fifth is a multiple of 5), so the number of zeros equals the number of factors 5 in $N!$.

How many factors 5 are there? Every multiple of 5 up to N contributes at least one; there are $\lfloor N/5 \rfloor$ of them, where $\lfloor x \rfloor$ means x rounded down to a whole number. Every multiple of 25 contributes a *second* factor 5, and there are $\lfloor N/25 \rfloor$ of those. Every multiple of 125 contributes a third, and so on. Adding the layers gives Legendre's formula for the prime 5 [2]:

$$Z(N) = \sum_{i \geq 1} \left\lfloor \frac{N}{5^i} \right\rfloor = \left\lfloor \frac{N}{5} \right\rfloor + \left\lfloor \frac{N}{25} \right\rfloor + \left\lfloor \frac{N}{125} \right\rfloor + \cdots \tag{5.1}$$

The sum looks infinite but is not: as soon as $5^i > N$ every further term is zero.

A worked example by hand, $N = 125$:

- multiples of 5 up to 125: $\lfloor 125/5 \rfloor = 25$;
- multiples of 25: $\lfloor 125/25 \rfloor = 5$;
- multiples of 125: $\lfloor 125/125 \rfloor = 1$;
- multiples of 625: $\lfloor 125/625 \rfloor = 0$, and every later term is 0 too.

So $Z(125) = 25 + 5 + 1 = 31$: the number 125! ends in exactly 31 zeros.

5.3.3 Why the program divides instead of multiplying

There are two ways to program Equation (5.1). The obvious one keeps a power $p = 5, 25, 125, \ldots$ and adds $\lfloor N/p \rfloor$ while $p \le N$. The other one never forms the power at all. It rests on a small fact about rounding down: dividing by 5 and rounding, then dividing by 5 again and rounding, is the same as dividing by 25 and rounding once,

$$\left\lfloor \frac{\lfloor N/5^{i} \rfloor}{5} \right\rfloor = \left\lfloor \frac{N}{5^{i+1}} \right\rfloor \tag{5.2}$$

(write $N = 5^{i+1}q + r$ with $0 \le r < 5^{i+1}$; both sides equal q). So the loop "divide N by 5, add the result, repeat until N is 0" produces exactly the terms of Equation (5.1), one after another, and every number it touches is at most N. Nothing can overflow, whatever the type of N. We will see in Section 5.5 that the other loop, harmless as it looks, walks off a cliff near the top of the `int` range.

How much work is it? The loop runs once per nonzero term, that is $\lfloor \log_5 N \rfloor + 1$ times; for $N = 10^9$ that is 13 divisions. With $T = 10^5$ values the whole file costs about

$$T \cdot (\lfloor \log_5 N \rfloor + 1) \le 10^5 \times 13 = 1.3 \times 10^6 \tag{5.3}$$

divisions, which is nothing. The answer is also small: each term is at most a fifth of the one before, so $Z(N) < N/5 + N/25 + \cdots = N/4$, at most 250 million, and it fits in a 32-bit `int`. We use `long long` anyway, so that the program stays right if the input is larger than promised.

5.3.4 The program

Listing 5.1 is the C++ program and **Listing 5.2** the Python one. Neither uses anything that depends on the platform: standard headers only, no pragmas, a type that is 64 bits everywhere, and the `printf` format `%lld` that matches it.

Listing 5.1 `fctrl.cpp`: the trailing zeros of $N!$, written to behave the same on every platform.

```cpp
// FCTRL (SPOJ): trailing zeros of N! for many N.
// Portable: standard headers only, no pragmas, no overflow for any N < 2^31.
#include <cstdio>

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        long long n;
        if (std::scanf("%lld", &n) != 1) break;
        long long zeros = 0;
        while (n > 0) {          // floor(N/5) + floor(N/25) + floor(N/125) + ...
            n /= 5;
            zeros += n;
        }
        std::printf("%lld\n", zeros);
    }
    return 0;
}
```

Listing 5.2 `fctrl.py`: the same count in Python, reading the whole input at once.

```python
# FCTRL (SPOJ): trailing zeros of N! for many N.
# Written for Python 3.8 or newer: no match statement, no new syntax.
import sys


def zeros(n):
    count = 0
    while n:
        n //= 5
        count += n
    return count


def main():
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    out = [str(zeros(int(x))) for x in data[1:1 + t]]
    sys.stdout.write("\n".join(out) + "\n")


if __name__ == "__main__":
    main()
```

Two details of the Python program deserve a word. It reads the entire input with `sys.stdin.buffer.read()` and splits it into tokens, which is the fastest simple way to read many numbers in Python (Chapter 4 measured the alternatives), and it collects all answers in a list and writes them with one call. It also avoids syntax newer than Python 3.8, such as the `match` statement, because a judge's Python can be older than yours; Section 5.8 comes back to this.

We feed both programs our own test file, with values chosen to hit each term of Equation (5.1) and the top of the range:

```
6
7
25
125
1000
999999999
1000000000
```

Compiled with `g++ -std=c++17 -O2 -Wall -Wextra -o fctrl fctrl.cpp` and run as `./fctrl < sample_fctrl.txt` (on Windows `.\fctrl.exe`), the C++ program prints the following; `python3 fctrl.py < sample_fctrl.txt` prints exactly the same lines.

```
1
6
31
249
249999989
249999998
```

The third line is our hand calculation for $N = 125$. The second is $Z(25) = 5 + 1 = 6$: 25 is the first number that contributes two fives. On a file of 100,000 random values up to 10^9, the largest input the problem allows, the C++ program took 0.013 seconds and the Python program 0.07 seconds, both far inside the limit.

5.4 What the Judge Does With Your Source

5.4.1 The compiler on the other side

When you submit, SPOJ's language menu asks you to choose among entries such as "C++14 (gcc 8.3)" and "C++ (gcc 8.3)"; solution archives that record the chosen entry show gcc 8.3 as the compiler behind them [3]. GCC 8 was released in 2018. Your own compiler is almost certainly newer: GCC 16.1 appeared on April 30, 2026, and one of its headline changes is that a plain `g++` command now compiles *C++20* by default, where earlier versions used C++17 [4]. Two lessons follow.

First, **always say which standard you mean**. Compile at home with `-std=c++17` (or whatever the judge's menu names), never with the default, because the default is a moving target. A feature that compiles silently under GCC 16's default C++20, such as `std::ranges::sort` or the `<=>` operator, is a compilation error on a judge that compiles C++14 or C++17.

Second, **older is not worse, only different**. Every program in this chapter and the next three compiles under C++14 as well as C++17; the check in Section 5.8 proves it for this chapter's programs. If you need a newer feature, check the judge's menu first.

The processors differ too. SPOJ runs submissions for most problems on a cluster it calls *Cube*, built from Intel Xeon E3-1220 v5 processors [5]. That processor, a 2015 design, supports the AVX2 vector instructions but not AVX-512 [6]. Your laptop may be older, newer or of a different kind altogether: an Apple laptop from 2020 onward has an ARM processor that runs none of these instructions. Keep that in mind for Section 5.7.

5.4.2 Asking the compiler what it is

The quickest way to learn the facts that matter is to ask the compiler. **Listing 5.3** prints the compiler, the language standard it is using, the sizes of the integer types, whether `char` is signed, and whether the 128-bit extension type `__int128` exists.

Listing 5.3 `platform_probe.cpp`: prints the facts about a compiler and a machine that decide whether a program behaves the same elsewhere.

```cpp
// Prints the facts about this compiler and machine that decide whether a
// program written here behaves the same on the judge.
#include <climits>
#include <cstdio>

int main() {
    std::printf("compiler     : ");
#if defined(__clang__)
    std::printf("clang %d.%d\n", __clang_major__, __clang_minor__);
#elif defined(__GNUC__)
    std::printf("g++ %d.%d\n", __GNUC__, __GNUC_MINOR__);
#elif defined(_MSC_VER)
    std::printf("MSVC %d\n", _MSC_VER);
#else
    std::printf("unknown\n");
#endif
    std::printf("__cplusplus  : %ldL\n", (long)__cplusplus);
    std::printf("sizeof int   : %zu\n", sizeof(int));
    std::printf("sizeof long  : %zu\n", sizeof(long));
    std::printf("sizeof llong : %zu\n", sizeof(long long));
    std::printf("sizeof ptr   : %zu\n", sizeof(void*));
    std::printf("char signed  : %s\n", CHAR_MIN < 0 ? "yes" : "no");
#ifdef __SIZEOF_INT128__
    std::printf("__int128     : available\n");
#else
    std::printf("__int128     : not available\n");
#endif
    std::printf("LONG_MAX     : %ld\n", LONG_MAX);
    return 0;
}
```

The `#if defined(...)` lines are evaluated by the *preprocessor* before compilation: each compiler defines its own identifying macros, such as `__GNUC__` for g++ and `__clang__` for clang, and only the matching branch is compiled. The macro `__cplusplus` holds the standard in use as a number: 201402 for C++14, 201703 for C++17, 202002 for C++20. On our Linux machine, compiled with `g++ -std=c++17 -O2`, the program printed:

```
compiler     : g++ 13.3
__cplusplus  : 201703L
sizeof int   : 4
sizeof long  : 8
sizeof llong : 8
sizeof ptr   : 8
char signed  : yes
__int128     : available
LONG_MAX     : 9223372036854775807
```

Listing 5.4 asks the same kind of questions of Python.

Listing 5.4 `platform_probe.py`: the Python version, implementation and limits that can differ between your machine and the judge.

```python
# The Python side of platform_probe.cpp: what could differ on the judge.
import platform
import struct
import sys

print("python       :", platform.python_version())
print("implementation:", platform.python_implementation())
print("pointer bits :", struct.calcsize("P") * 8)
print("int max      : unlimited (sys.maxsize = %d)" % sys.maxsize)
print("recursion    :", sys.getrecursionlimit())
print("int->str cap :", getattr(sys, "get_int_max_str_digits", lambda: "none")())
```

```
python       : 3.11.15
implementation: CPython
pointer bits : 64
int max      : unlimited (sys.maxsize = 9223372036854775807)
recursion    : 1000
int->str cap : 4300
```

The last line is easy to overlook and important: since Python 3.11, converting between `int` and `str` refuses numbers with more than 4300 digits and raises `ValueError`, a protection against a denial-of-service attack [7]. A program that prints 1000! with `print` works on Python 3.10 and fails on 3.11 and later unless it raises the limit with `sys.set_int_max_str_digits`. Chapter 8 meets this limit again in a problem with a million-digit number.

5.4.3 The sizes of the integer types

The C++ standard fixes only minimum sizes for its integer types; the actual sizes are chosen by each platform's *data model* [8]. **Table 5.2** shows the three combinations you are likely to meet.

Table 5.2 Integer sizes in the two 64-bit data models; the bold row is the one that differs.

Type	Linux and macOS, 64-bit (LP64)	Windows, 64-bit (LLP64)	What to use for
`int`	32 bits	32 bits	values below about 2.1×10^9
`long`	**64 bits**	**32 bits**	avoid in contest code
`long long`	64 bits	64 bits	values up to about 9.2×10^{18}
pointer, `size_t`	64 bits	64 bits	sizes and indexes of containers
`__int128`	available in g++ and clang	not in MSVC	intermediate products up to 10^{38}

The row for `long` is the reason for Sinta's lost evening in Section 5.1. Microsoft documents that `long` is 4 bytes in its compilers on every Windows platform [9], and MinGW-w64, the g++ for Windows that Chapter 1 installs, follows the same data model so that it can call Windows itself. The rule for contest code is therefore simple: write `int` when the value is below two billion, `long long` when it may not be, and never plain `long`.

5.5 Flags That Find Bugs

5.5.1 Warnings

A warning is the compiler saying "this is legal, but it is probably not what you meant". The flags `-Wall` and `-Wextra` switch on the warnings that experienced programmers want to see [10]. The first version of **Listing 5.1** ignored the value returned by `scanf`; compiled with `-Wall -Wextra`, g++ warned about it:

```
fctrl.cpp:10:19: warning: ignoring return value of 'int scanf(const char*, ...)'
declared with attribute 'warn_unused_result' [-Wunused-result]
```

That is not a disaster in a contest, since the judge's input is well formed, but checking the return value costs nothing and makes the program stop cleanly on a truncated file instead of looping on stale values. The version printed in **Listing 5.1** checks it, and compiles without a single warning. Make that your habit: compile with `-Wall -Wextra` at home and treat every warning as a question you must answer.

5.5.2 Sanitizers

Some bugs cannot be seen by reading the text; they happen only while the program runs, with particular values. A **sanitizer** is a checking mode in which the compiler adds tests to the machine code, so that the program stops with a report at the exact line where the bug happens [10]. Two are worth knowing:

- `-fsanitize=undefined` (UBSan) reports undefined behavior such as signed overflow, division by zero or shifting by too many bits;
- `-fsanitize=address` (ASan) reports reads and writes outside arrays, use of freed memory and similar memory errors.

Sanitized programs run a few times slower and use more memory, so you use them at home, never on the judge. They work with g++ and clang on Linux, with clang on macOS, and under WSL on Windows; the MinGW-w64 compiler does not ship them.

To see UBSan at work, **Listing 5.5** computes Equation (5.1) the obvious way, with a growing power of 5.

Listing 5.5 `fctrl_mulloop.cpp`: the same formula with a growing power of 5; correct inside the constraints, undefined behavior just outside them.

```cpp
// The same count, written with a growing power of 5. Looks innocent;
// for N near 2^31 the power overflows a 32-bit int (undefined behavior).
#include <cstdio>

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        int n;
        if (std::scanf("%d", &n) != 1) break;
        int zeros = 0;
        for (int p = 5; p <= n; p *= 5)   // p *= 5 can pass INT_MAX
            zeros += n / p;
        std::printf("%d\n", zeros);
    }
    return 0;
}
```

On the sample file it prints exactly the same six lines as **Listing 5.1**:

```
1
6
31
249
249999989
249999998
```

Within FCTRL's constraints it is even correct: for $N \le 10^9$ the loop stops at $p = 5^{13} = 1\,220\,703\,125$, which is larger than N and still below the `int` limit $2^{31} - 1 = 2\,147\,483\,647$. But it has no safety margin. For any N from 5^{13} up to $2^{31} - 1$, the step `p *= 5` must compute $5^{14} = 6\,103\,515\,625$, which does not fit in an `int`. That is a signed overflow, and signed overflow is undefined behavior [11]. We fed the program a two-line file with $N = 10^9$ and $N = 2^{31} - 1$. Compiled normally, it printed nothing and ran until the five-second timeout of our test script killed it: the overflowed power became a negative number, still "at most N", and the loop never ended. A wrong constant in the input specification, a problem that says $N < 2^{31}$ instead of $N \le 10^9$, would turn this program into *Time limit exceeded*. Compiled with UBSan,

```
g++ -std=c++17 -O2 -fsanitize=undefined -fno-sanitize-recover=all \
    -o mul_san fctrl_mulloop.cpp
```

the same input stops immediately, with the line, the column and the values:

```
fctrl_mulloop.cpp:12:35: runtime error: signed integer overflow:
1220703125 * 5 cannot be represented in type 'int'
```

(The report is one line on the screen; it is wrapped here to fit the page.) Notice also that the answer for $N = 10^9$, which the program computed before the overflow, is missing from the report: the program was stopped before its output buffer was written. When a crashing program prints less than you expect, it has not necessarily stopped earlier than you think.

5.5.3 The optimizer assumes that undefined behavior never happens

Why did the broken loop run forever instead of stopping? Because the compiler is allowed to assume that a correct program never overflows, and it uses that assumption to make code faster. **Listing 5.6** shows the effect in the smallest program we could write.

Listing 5.6 `ub_check.cpp`: the loop condition `i > 0` depends on overflow, so the number of steps depends on the optimization level.

```
// Signed overflow is undefined behavior: the optimizer may assume it never
// happens, so the same source can print different numbers at -O0 and -O2.
#include <climits>
#include <cstdio>

int main() {
    volatile int start = INT_MAX - 2500000;  // volatile: value unknown at compile
    int steps = 0;
    for (int i = start; i > 0 && steps < 100; i += 1000000)
        steps++;              // i > 0 "cannot fail" if overflow never happens
    std::printf("steps = %d\n", steps);
    unsigned int u = UINT_MAX;
    u = u + 1;                // unsigned arithmetic wraps: well defined
    std::printf("UINT_MAX + 1 = %u\n", u);
    return 0;
}
```

The variable `start` is declared `volatile`, which forces the compiler to read it at run time, so that it cannot compute the answer in advance. The loop starts 2.5 million below the largest `int` and adds one million per step. Reasoning with real integers, i stays positive forever, and the loop is stopped only by `steps < 100`. Reasoning with 32-bit arithmetic, the third addition wraps around to a negative number and the loop stops after 3 steps. Compiled with `-O2`, the program printed:

```
steps = 100
UINT_MAX + 1 = 0
```

Compiled with `-O0`, no optimization, the same source printed `steps = 3`. We tried both compilers:

Table 5.3 The output of `ub_check.cpp` under different compilers and optimization levels.

Compiler and flags	`steps` printed
g++ 13.3, `-O0`	3
g++ 13.3, `-O1`	100
g++ 13.3, `-O2`	100
clang 18.1, `-O0`	3
clang 18.1, `-O2`	100
g++ 13.3, `-O2 -fsanitize=undefined`	stops with "signed integer overflow: 2146983647 + 1000000"

Table 5.3 shows the heart of the matter. At `-O1` and above, both compilers reason "i starts positive and only grows, and overflow cannot happen in a correct program, so `i > 0` is always true", delete the test, and run the loop 100 times. Neither compiler is wrong; the program is. The second half of **Listing 5.6** shows the contrast: *unsigned* arithmetic is defined to wrap around modulo 2^{32}, so `UINT_MAX + 1` is 0 on every compiler at every level.

For you, the consequence is practical. A program with undefined behavior can pass your tests at `-O0`, the default of many editors' "Run" buttons, and fail on the judge at `-O2`, or the other way round. Test with the flags the judge uses, and run your stress tests (Chapter 8) once with the sanitizers switched on.

5.5.4 The debug mode of the standard library

One more checking mode is specific to g++: compiling with `-D_GLIBCXX_DEBUG` replaces the containers of the standard library with versions that check every index and iterator [12]. An out-of-range `v[i]` then stops the program with a clear message instead of silently reading a neighbor's memory. Chapter 8 uses it on a real bug; for now, add it to the list of switches you use at home and never on the judge.

5.6 Flags That Make Code Faster

5.6.1 Optimization levels

Without instructions, g++ translates each line of source as literally as it can, so that a debugger can follow the program line by line. That is optimization level `-O0`. The higher levels let the compiler keep values in registers instead of memory, remove computations whose results are never used, move work out of loops, replace calls by the body of the function (*inlining*), and, from `-O2` on in recent GCC versions, process several array elements with one *vector* instruction (*vectorization*) [13]. `-O3` adds more aggressive transformations, and `-march=native` tells the compiler to use every instruction of the processor it is running on.

Judges compile with optimization; most use `-O2`. To see how much the level matters, we timed **Listing 5.7**, a fixed amount of simple work: for each of 20,000 thresholds, count how many of 20,000 values are below it, 4×10^8 comparisons in all.

Listing 5.7 `flags_bench.cpp`: 4×10^8 comparisons whose speed depends only on the compiler flags.

```cpp
// A fixed amount of work whose speed depends on the compiler flags:
// for each of Q thresholds, count the array values below it (Q * N steps).
// The checksum is printed to stdout; the time goes to stderr.
#include <chrono>
#include <cstdint>
#include <cstdio>
#include <vector>

int main() {
    const int N = 20000, Q = 20000;
    std::vector<int> a(N), q(Q);
    std::uint32_t s = 12345;                  // tiny generator, same everywhere
    for (int& x : a) { s = s * 1103515245u + 12345u; x = (int)(s >> 8) % 1000000; }
    for (int& x : q) { s = s * 1103515245u + 12345u; x = (int)(s >> 8) % 1000000; }
    auto start = std::chrono::steady_clock::now();
    long long checksum = 0;
    for (int j = 0; j < Q; j++) {
        int below = 0;
        for (int i = 0; i < N; i++) below += a[i] < q[j];  // no branch: vectorizes
        checksum += below;
    }
    auto stop = std::chrono::steady_clock::now();
    std::printf("checksum %lld\n", checksum);
    std::fprintf(stderr, "time %.3f s\n",
                 std::chrono::duration<double>(stop - start).count());
    return 0;
}
```

Two details make the benchmark honest. The numbers come from a tiny random generator written out in the program, so every compiler on every platform sees exactly the same data; and the program prints a checksum, the total of all counts, so we can confirm that every build computes the same thing. The time goes to the error stream, `stderr`, so that the normal output stays identical:

```
checksum 200865372
```

Note also the line `below += a[i] < q[j];`. The comparison is 1 or 0, and adding it avoids an `if` inside the inner loop; a loop without branches is exactly what vector instructions are good at.

5.6.2 What we measured

We compiled **Listing 5.7** with nine different settings and took the fastest of three runs of each, on an otherwise idle machine; a first series, run while other jobs were using the processor, gave times about twice as long in every row, with the same ratios. **Table 5.4** and **Figure 5.1** show the result; the checksum was 200865372 every time.

Table 5.4 Time of `flags_bench.cpp` under nine settings (fastest of three runs, g++ 13.3, Xeon at 2.1 GHz).

Flags	Time (s)	Runs on any x86-64 processor?
`-O0`	1.692	yes
`-O1`	0.268	yes
`-O2`	0.044	yes
`-O3`	0.041	yes
`-O3 -funroll-loops`	0.033	yes
`-O2 -mavx2`	0.017	only with AVX2
`-O2 -march=native`	0.018	only on this processor model
`-O0` + both pragmas of **Listing 5.8**	1.127	only with AVX2
`-O2` + both pragmas of **Listing 5.8**	0.015	only with AVX2

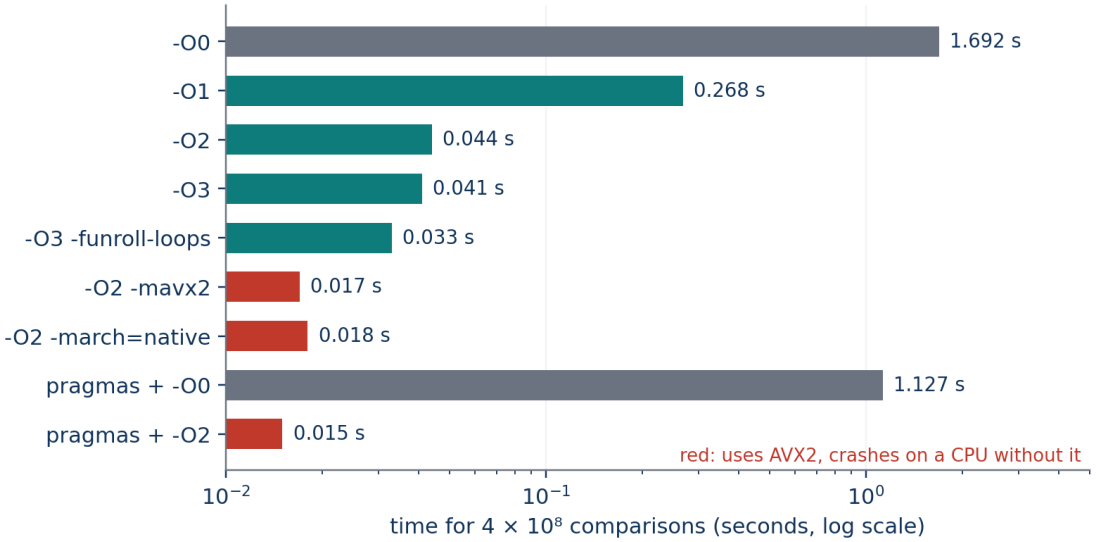


Figure 5.1 The same 4×10^8 comparisons under nine flag settings, on a logarithmic scale; red bars use AVX2 instructions.

Four lessons stand out.

1. **The optimization level dominates everything else.** From `-O0` to `-O2` the program became 38 times faster, without a single change to the source. Timing a program built without optimization says nothing about how it will do on the judge. If your editor's run button compiles with `-O0`, a solution that "takes two seconds" at home may take a twentieth of a second on the judge, and you may throw away a correct idea for nothing.
2. **`-O3` is not a magic switch.** Here it gained 7% over `-O2`; on other programs it gains nothing or loses a little. Adding unrolling gained a further fifth. These are small effects compared with the choice of algorithm.
3. **Wider vector instructions pay off on loops like this one**, a factor of two and a half over `-O2`, but only on processors that have them.
4. **A pragma does not replace the command-line level.** The pragmas of **Listing 5.8** with an `-O0` command line were only 1.5 times faster than plain `-O0`, because the parts of the standard library defined before the pragma were still compiled without optimization.

5.6.3 Optimization from inside the source

Because you cannot change the judge's flags, GCC lets you ask for optimization from inside the file. `#pragma GCC optimize("O3,unroll-loops")` sets optimization options, and `#pragma GCC target("avx2")` allows instructions of a processor extension, both for the functions defined after the line [14]. **Listing 5.8** is **Listing 5.7** with the two pragmas added.

Listing 5.8 `flags_bench_pragma.cpp`: the benchmark with an optimize pragma and a target pragma, placed after the includes.

```
// flags_bench.cpp with two pragmas: the optimize pragma raises the
// optimization level, the target pragma allows AVX2 instructions.
// A CPU without AVX2 stops this program with "Illegal instruction".
// The pragmas come AFTER the includes: placed first, g++ 13 refuses to
// inline the library's helpers ("target specific option mismatch").
#include <chrono>
#include <cstdint>
#include <cstdio>
#include <vector>
#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("avx2")

int main() {
    const int N = 20000, Q = 20000;
    std::vector<int> a(N), q(Q);
    std::uint32_t s = 12345;
    for (int& x : a) { s = s * 1103515245u + 12345u; x = (int)(s >> 8) % 1000000; }
    for (int& x : q) { s = s * 1103515245u + 12345u; x = (int)(s >> 8) % 1000000; }
    auto start = std::chrono::steady_clock::now();
    long long checksum = 0;
    for (int j = 0; j < Q; j++) {
        int below = 0;
        for (int i = 0; i < N; i++) below += a[i] < q[j];
        checksum += below;
    }
    auto stop = std::chrono::steady_clock::now();
    std::printf("checksum %lld\n", checksum);
    std::fprintf(stderr, "time %.3f s\n",
                 std::chrono::duration<double>(stop - start).count());
    return 0;
}
```

```
checksum 200865372
```

The comment at the top of the listing records something we learned the hard way. Blogs usually put the pragmas on the first lines of the file, before the includes. We did that first, and g++ 13.3 refused to compile the program:

```
error: inlining failed in call to 'always_inline' 'std::allocator<...>::~allocator()
noexcept [with _Tp = int]': target specific option mismatch
```

With the target pragma first, the library's own small functions were compiled for AVX2 while other parts that call them were not, and the compiler will not inline code compiled for a larger instruction set into code compiled for a smaller one. Whether this error appears depends on the compiler version and on which headers you include, which is itself a lesson: a pragma copied from someone else's template can turn an accepted program into a compilation error on a different compiler. Moving the pragmas after the includes, as in **Listing 5.8**, applies them to your own functions only and compiles cleanly with every compiler we tried.

5.7 The Pragma Risk Ladder

A pragma is a trade: some speed for some risk. **Figure 5.2** arranges the common choices as a ladder, from the bottom rung, which is always safe, to the top, which is a gamble.

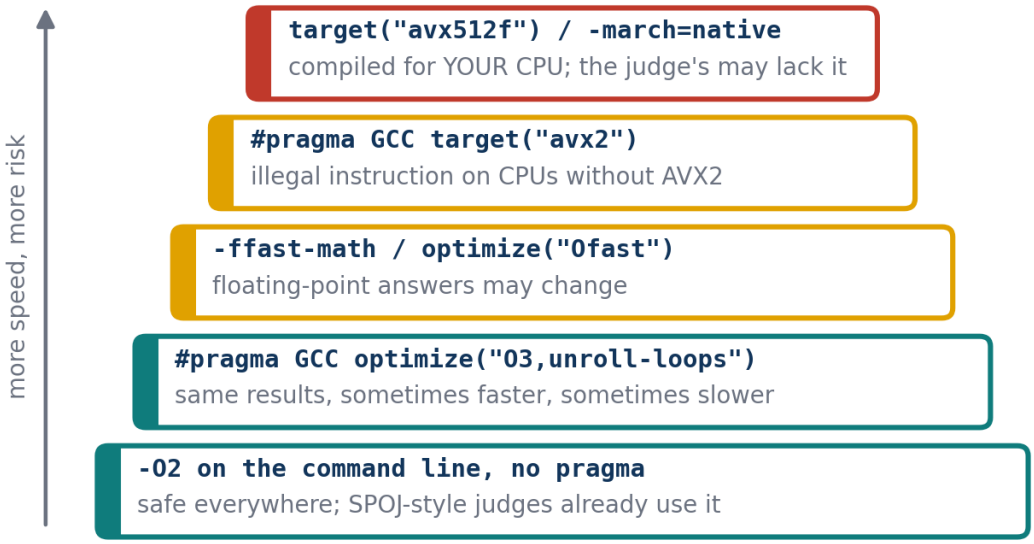


Figure 5.2 The pragma risk ladder: each rung up buys speed on some machines and adds a way to fail on others.

Rung 1: the judge's `-O2`, nothing else. This is what every accepted solution in this book uses. It is portable by definition.

Rung 2: `#pragma GCC optimize("O3,unroll-loops")`. It changes only *how* the compiler translates, not which instructions it may use, so a correct program stays correct and runs on every processor. The gain is uncertain, as **Table 5.4** shows, and the GCC manual describes these options as meant for tuning individual functions, not as a general switch [14]. Harmless, but rarely the difference between accepted and not.

Rung 3: `-ffast-math` or `optimize("Ofast")`. These allow the compiler to reorder floating-point arithmetic as if it were exact real arithmetic, and to assume that no value is ever infinite or "not a number" [13]. For integer code they change nothing. For floating-point code they can change the last digits of a result, and a check such as `if (x != x)` for "not a number" may be removed altogether. In a geometry problem that compares distances with a tolerance of 10^{-9}, that is a real risk of *Wrong answer*.

Rung 4: `#pragma GCC target("avx2")` and similar. The compiler may now emit AVX2 instructions. On a processor without AVX2 the program is stopped by the operating system with the signal SIGILL, "illegal instruction", which a judge reports as a runtime error. SPOJ's Cube processors have AVX2 [6], so on SPOJ this rung happens to work; on another judge, or on an older cluster, it may not.

Rung 5: `target("avx512f")` or `-march=native`. AVX-512 is absent from the Xeon E3-1220 v5 [6], and `-march=native` means "whatever the machine I am compiling on has", which is meaningless when the machine that compiles is not the one you tested on. This is exactly Bima's failure in Section 5.1: the pragma made the program faster on his laptop and killed it on the judge. On an

Apple laptop with an ARM processor, x86 target pragmas do not even compile, which at least fails early.

The ladder suggests a rule you can apply in seconds. **Climb only when the algorithm is right and the measured time is close to the limit, and never above rung 2 without knowing the judge's processor.** A solution that needs rung 4 to pass is usually a solution with the wrong complexity, and Chapter 3's budget table is a better place to look than a pragma.

5.8 A Portability Checklist

5.8.1 C++

The failures of Section 5.1 belong to a short list that covers almost every portability problem in contest code. **Table 5.5** collects it.

Table 5.5 Features that break when a C++ program moves to another platform, and what to write instead.

Feature	Where it breaks	Portable replacement
`#include <bits/stdc++.h>`	clang with libc++ (the default on macOS) [15], MSVC	include the standard headers you use
`long` for values above 2^{31}	Windows (32-bit `long`)	`long long`, printed with `%lld`
`__int128`	MSVC; printing needs your own code everywhere	split the computation, or use it only where g++/clang are guaranteed
variable-length arrays `int a[n];`	MSVC; large `n` overflows the stack everywhere	`std::vector<int> a(n);`
`gets`, `itoa`, `strrev`, `conio.h`	removed from C++14 or never standard	`std::getline`, `std::to_string`, `std::reverse`
`char` holding bytes above 127	ARM Linux, where `char` is unsigned	`unsigned char` or `int`
C++20 features (`<=>`, ranges, `std::format`)	judges compiling C++14 or C++17	the C++17 equivalents, or check the menu
deep recursion	Windows (1 MB stack), some judges	an explicit stack (Chapter 6)
signed overflow, uninitialized variables	any compiler at any level	64-bit types, initialize everything, test with sanitizers

The first row deserves a comment, because `bits/stdc++.h` appears in countless contest templates. It is an internal header of GCC's standard library, libstdc++, that includes everything, and many published contest templates start with it [16]; it is convenient and slows compilation a little. The judge uses g++, so it works on SPOJ. It fails on a Mac, whose default compiler uses LLVM's libc++, and on Microsoft's compiler. If you share code with others, or use a Mac, write the individual includes; it costs a few lines.

5.8.2 Python

Python programs are more portable than C++ ones, since the interpreter hides the processor, but three differences still matter.

- **The version.** A judge's Python can be several versions behind yours. Syntax such as the `match` statement (3.10), exception groups (3.11) or the new generic syntax (3.12) is a syntax error on an older interpreter, which the judge reports as a runtime error. When in doubt, write Python that 3.8 accepts, as **Listing 5.2** does.

- **The limits.** The recursion limit is 1000 by default, and since 3.11 integer-to-string conversion is capped at 4300 digits [7]. Both are settings, not laws, but the program must change them explicitly.
- **The implementation.** Some judges offer PyPy, a faster implementation of Python, beside CPython. PyPy runs loops many times faster, but has its own memory behavior and lags CPython in version. Test with the one you submit to.

5.8.3 A script that checks before you submit

Reading a checklist is good; running one is better. **Listing 5.9** compiles every C++ program in the chapter's `cpp` folder five times: as C++14, as C++17 with warnings turned into errors, as C++20, with both sanitizers, and with clang.

Listing 5.9 `portability_check.py`: compiles every program under several standards, compilers and checking modes and reports the failures.

```
# Compile every program in ../cpp under several compilers, standards and
# checking modes, and report which combinations fail. Run it before submitting.
import glob
import os
import shutil
import subprocess

MODES = [
    ("g++", ["-std=c++14", "-O2"]),
    ("g++", ["-std=c++17", "-O2", "-Wall", "-Wextra", "-Werror"]),
    ("g++", ["-std=c++20", "-O2"]),
    ("g++", ["-std=c++17", "-O1", "-fsanitize=address,undefined"]),
    ("clang++", ["-std=c++17", "-O2", "-Wall", "-Wextra"]),
]

here = os.path.dirname(os.path.abspath(__file__))
sources = sorted(glob.glob(os.path.join(here, "..", "cpp", "*.cpp")))
out = os.path.join(here, "build")
os.makedirs(out, exist_ok=True)
for compiler, flags in MODES:
    if shutil.which(compiler) is None:
        print("%-9s %-44s skipped (not installed)" % (compiler, " ".join(flags)))
        continue
    bad = []
    for src in sources:
        exe = os.path.join(out, "probe")
        r = subprocess.run([compiler] + flags + ["-o", exe, src],
                           capture_output=True, text=True)
        if r.returncode != 0:
            bad.append(os.path.basename(src))
    status = "FAIL: " + ", ".join(bad) if bad else "all %d compile" % len(sources)
    print("%-9s %-44s %s" % (compiler, " ".join(flags), status))
```

Run from the `tests` folder with `python3 portability_check.py`, it printed:

```
g++       -std=c++14 -O2                                all 6 compile
g++       -std=c++17 -O2 -Wall -Wextra -Werror          all 6 compile
g++       -std=c++20 -O2                                all 6 compile
g++       -std=c++17 -O1 -fsanitize=address,undefined   all 6 compile
clang++   -std=c++17 -O2 -Wall -Wextra                  all 6 compile
```

Every program of the chapter compiles in every mode. The same script found the pragma error of Section 5.6.3 before we moved the pragmas, and it flagged the `scanf` warning of Section 5.5.1 in the `-Werror` row. On Windows with MSYS2 the sanitizer row fails, because MinGW-w64 has no

sanitizers, and the clang row is skipped unless clang is installed; the other rows work as they do here.

5.9 Testing It

The formula of Section 5.3 is proved, but a proof is about the formula, not about the program. The program could still divide by the wrong number, print the wrong type, or read the input wrongly. Chapter 8 develops the method we use for every problem in this book; here is a first taste.

The idea is to write a second program that is *obviously* right, however slow it is, and to compare the two on many random inputs. For FCTRL the obvious program computes $N!$ exactly, with Python's unlimited integers, and counts the zeros at the end of its decimal form, character by character. **Listing 5.10** is that program.

Listing 5.10 `brute_fctrl.py`: the brute force for FCTRL builds $N!$ and counts the zeros; only small N are feasible.

```
# Brute force for FCTRL: build N! exactly and count its trailing zeros.
# Only usable for small N, which is the point: it is obviously right.
import sys

data = sys.stdin.read().split()
t = int(data[0])
for x in data[1:1 + t]:
    f = 1
    for i in range(2, int(x) + 1):
        f *= i
    s = str(f)
    print(len(s) - len(s.rstrip("0")))
```

It can only handle small N (computing $10^9!$ is out of the question), so the generator `gen_fctrl.py` produces between one and eight values up to 400. For large N we use a second opinion, `brute_fctrl_big.py`, which evaluates Equation (5.1) with growing powers of 5 in Python's exact integers, a different computation from the division loop of **Listing 5.1**. The harness `stress.py`, listed and explained in Chapter 8, runs generator, brute force and candidate for 300 random seeds:

```
python3 stress.py "python3 gen_fctrl.py" "python3 brute_fctrl.py" \
    build/fctrl 300
python3 stress.py "python3 gen_fctrl.py big" "python3 brute_fctrl_big.py" \
    build/fctrl 300
```

```
OK: 300 random tests agree
OK: 300 random tests agree
```

The Python program agreed on all 300 tests as well. The "big" generator deliberately includes $N = 2^{31} - 1$, just outside the constraints, and running it against `fctrl_mulloop` reproduced the endless loop of Section 5.5.2 on the third seed:

```
MISMATCH on seed 3
input:
4
2147483647
244140625
671862058
650257552
expected: 536870902 61035156 167965506 162564382
received: TIMEOUT (more than 5 s)
```

That is the kind of result a stress test is for: a program that passed every hand-made test, with a hidden cliff one input away.

5.10 Using It for Real

Everything in this chapter becomes a habit once it is written into your tools. Three small investments make it automatic.

A compile command you always use. Decide once which flags you compile with at home and keep them in a script, an alias or your editor's settings. A good everyday command mirrors the judge and adds the warnings:

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

and a good "suspicious" command adds the checks:

```
g++ -std=c++17 -O1 -g -Wall -Wextra -fsanitize=address,undefined \
    -D_GLIBCXX_DEBUG -o prog prog.cpp
```

In VS Code, put both in the `.vscode/tasks.json` file of your contest folder as two build tasks, so that **Terminal > Run Build Task** runs the everyday one; the C/C++ extension's default task compiles with `-g` only, which is `-O0`, and would mislead every timing you make.

A template without surprises. If you keep a file you copy at the start of every problem, make it portable: individual includes, `long long` rather than `long`, no target pragmas, fast input and output from Chapter 4. Anything more exotic goes in on purpose, for one problem, with a comment saying why.

A last check before submitting. For a problem with tight limits or unusual types, run `portability_check.py` or at least the sanitizer build on your stress tests, and time the largest input with the everyday command. The whole routine takes a minute, and it is the minute that turns "it worked at home" into "it works".

The same thinking applies outside contests. A shop owner in Sidoarjo who writes a small program to total the day's sales on her Windows laptop, and later runs it on a cheap Linux server, meets exactly the `long` problem of **Table 5.2** if the total in rupiah ever exceeds two billion, which a busy month does. Writing `long long` from the start, and testing on both machines, is the whole cure.

5.11 Common Mistakes, and How to Avoid Them

- **Timing a program built without optimization.** An `-O0` build can be nearly 40 times slower than the judge's `-O2` build (**Table 5.4**). Always time with the judge's flags.
- **Relying on the compiler's default standard.** GCC 16 compiles C++20 by default, older compilers C++14 or C++17. Pass `-std=` explicitly, matching the judge's menu.
- **Using `long` for 64-bit values.** It is 32 bits on Windows. Use `long long` and `%lld`.
- **Copying a pragma template without understanding it.** Target pragmas can crash on the judge's processor, and pragmas placed before the includes can stop compilation. Climb the ladder of **Figure 5.2** one rung at a time, and only with a measurement that justifies it.
- **Believing a program is right because it gives the right answers at one optimization level.** Undefined behavior can change with the level (**Table 5.3**). Run the sanitizers at least once.
- **Growing a power until it passes N.** `for (p = 5; p <= n; p *= 5)` overflows when n is near the type's limit. Divide N instead, as Equation (5.2) allows.

- **Writing Python for your own version only.** New syntax is a runtime error on an older judge; the 4300-digit conversion limit is an error on a newer one. Know which Python you submit to.

5.12 Chapter Summary

- Your program is compiled and run on the judge's machine, with the judge's compiler, standard and flags. Only the source text is yours.
- Differences come in three kinds: the language accepted (standard, headers, extensions), the sizes of types (`long` is 32 bits on Windows), and undefined behavior, which can give different results with different compilers and levels.
- FCTRL counts the factors 5 of $N!$ with Legendre's formula (Equation (5.1)); dividing N repeatedly (Equation (5.2)) evaluates it with no risk of overflow, in about 10^6 operations for the largest file (Equation (5.3)).
- `-Wall -Wextra` shows suspicious code, `-fsanitize=undefined,address` catches overflow and memory errors at run time, `-D_GLIBCXX_DEBUG` checks container indexes. Use them at home, not on the judge.
- Optimization level matters enormously: 38 times between `-O0` and `-O2` on our benchmark. `-O3`, unrolling and pragmas change far less, and target pragmas can crash on a processor without the instructions they allow.
- A short checklist (**Table 5.5**) and a script that compiles under several modes (**Listing 5.9**) catch nearly every portability problem before the judge does.

5.13 Exercises

1. By hand, compute $Z(1000)$ with Equation (5.1), and then the smallest N with $Z(N) = 1000$. Check both with `fctrl.cpp`.
2. Is there an N with $Z(N) = 5$? Explain which values Z skips and why.
3. Rewrite `fctrl_mulloop.cpp` so that it keeps the growing power but cannot overflow for any `int` value of N, without using a 64-bit type. Explain why your loop is safe.
4. Compile `ub_check.cpp` with `-O2 -fwrapv` and run it. What does `-fwrapv` change, and why is it not a good reason to rely on signed overflow in contest code?
5. A friend's template begins with `#pragma GCC target("avx2,bmi,bmi2,popcnt,lzcnt")` and `#include <bits/stdc++.h>`. List every platform from this chapter on which a correct program with this template can fail, and say how it fails on each.
6. The Python program **Listing 5.2** must print 100000! exactly (all its digits) on a judge running Python 3.12. Write the lines you would add, and explain what happens without them.

5.14 Solutions to the Exercises

1. The terms are $\lfloor 1000/5 \rfloor = 200$, $\lfloor 1000/25 \rfloor = 40$, $\lfloor 1000/125 \rfloor = 8$, $\lfloor 1000/625 \rfloor = 1$ and then 0, so $Z(1000) = 249$, which is also the fourth line of the sample output. For the second part, Z grows by roughly $N/4$, so try N near 4000: $Z(4000) = 800 + 160 + 32 + 6 + 1 = 999$. Z changes only at multiples of 5, and $Z(4005) = 801 + 160 + 32 + 6 + 1 = 1000$, so the smallest N is 4005. The program prints 999 for 4004 and 1000 for 4005.

2. No. $Z(24) = 4$ and $Z(25) = 6$: at $N = 25$ the number 25 brings two factors 5 at once, so Z jumps by 2 and the value 5 is skipped. In general Z jumps by k at a multiple of 5^k that is not a multiple of 5^{k+1}, and skips $k - 1$ values there.

3. Test before multiplying: continue while $p \le N/5$, that is, while multiplying by 5 cannot exceed N.

```
int zeros = 0;
for (int p = 5; ; p *= 5) {
    zeros += n / p;
    if (p > n / 5) break;   // 5 * p would exceed n: stop before computing it
}
```

If $p \le \lfloor N/5 \rfloor$ then $5p \le N \le 2^{31} - 1$, so the next multiplication fits; otherwise $5p > N$ and every later term is 0. For $N < 5$ the first term is 0 and the loop stops at once, which is correct.

4. `-fwrapv` tells g++ to define signed overflow as wrapping around, like unsigned arithmetic. The optimizer may then no longer delete the test `i > 0`, and the program prints `steps = 3` at every level. It is not a reason to rely on overflow, because the judge does not compile with `-fwrapv`, and because a wrapped value is still a wrong value: in FCTRL a wrapped power is negative, and the loop still misbehaves.

5. On macOS with Apple clang, `bits/stdc++.h` does not exist: compilation error. On an Apple laptop with an ARM processor, the x86 target names are unknown: compilation error. With some g++ versions and headers, the target pragma before the includes gives the inlining error of Section 5.6.3: compilation error. On a judge whose processor lacks one of the listed extensions, the program may be stopped with SIGILL: runtime error, but only on tests that reach the code that uses those instructions, which makes it look like a data-dependent bug. On SPOJ's Cube machines the listed extensions are present, so the template happens to work there.

6. Add `import sys` and, before printing, `sys.set_int_max_str_digits(0)` (0 means no limit), guarded for older versions:

```
if hasattr(sys, "set_int_max_str_digits"):
    sys.set_int_max_str_digits(0)
```

100000! has 456,574 digits. Without the call, `str()` or `print()` of the number raises `ValueError: Exceeds the limit (4300 digits) for integer string conversion`, and the judge reports a runtime error. The `hasattr` guard keeps the program working on Python 3.10 and older, where the function does not exist and there is no limit.

References

[1] Sphere Online Judge, "FCTRL - Factorial." [Online]. Available: https://www.spoj.com/problems/FCTRL/ (accessed Sep. 2026).

[2] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, 6th ed., revised by D. R. Heath-Brown and J. H. Silverman. Oxford, UK: Oxford University Press, 2008.

[3] hkktr, "polski-spoj: solutions for POLSKI SPOJ problems," README, GitHub. [Online]. Available: https://github.com/hkktr/polski-spoj (accessed Sep. 2026).

[4] GCC Steering Committee, "GCC 16.1 Released," gcc-announce mailing list, Apr. 30, 2026. [Online]. Available: https://gcc.gnu.org/pipermail/gcc-announce/2026/000190.html

[5] Wikipedia, "SPOJ." [Online]. Available: https://en.wikipedia.org/wiki/SPOJ (accessed Sep. 2026).

[6] Intel Corporation, "Intel Xeon Processor E3-1220 v5 (8M Cache, 3.00 GHz): Product Specifications." [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/88172/intel-xeon-processor-e31220-v5-8m-cache-3-00-ghz/specifications.html (accessed Sep. 2026).

[7] Python Software Foundation, "Built-in Types: Integer string conversion length limitation," *The Python Standard Library*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/stdtypes.html#int-max-str-digits

[8] cppreference.com, "Fundamental types." [Online]. Available: https://en.cppreference.com/w/cpp/language/types (accessed Sep. 2026).

[9] Microsoft, "Data Type Ranges," *Microsoft C++ Language Reference*, Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/cpp/cpp/data-type-ranges (accessed Sep. 2026).

[10] Free Software Foundation, "Options to Request or Suppress Warnings" and "Program Instrumentation Options," *Using the GNU Compiler Collection (GCC)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html (accessed Sep. 2026).

[11] cppreference.com, "Arithmetic operators: Overflows." [Online]. Available: https://en.cppreference.com/w/cpp/language/operator_arithmetic (accessed Sep. 2026).

[12] Free Software Foundation, "Debug Mode: Using," *The GNU C++ Library Manual*. [Online]. Available: https://gcc.gnu.org/onlinedocs/libstdc++/manual/debug_mode_using.html (accessed Sep. 2026).

[13] Free Software Foundation, "Options That Control Optimization," *Using the GNU Compiler Collection (GCC)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html (accessed Sep. 2026).

[14] Free Software Foundation, "Function Specific Option Pragmas," *Using the GNU Compiler Collection (GCC)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Function-Specific-Option-Pragmas.html (accessed Sep. 2026).

[15] LLVM Project, "libc++ C++ Standard Library." [Online]. Available: https://libcxx.llvm.org/ (accessed Sep. 2026).

[16] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Singapore: Lulu, 2020.

Chapter 6

Memory and Data-Structure Choices Under a Byte Budget

Every judge gives your program two budgets: seconds and bytes. Chapter 3 taught you to spend the first; this chapter is about the second. By the end of it you will be able to estimate, before writing any code, how many bytes a data structure will need; say what one element really costs in the common C++ containers and Python types, from measurements rather than guesses; solve SPOJ's PRIME1 with a segmented sieve that needs a hundred kilobytes where the obvious sieve needs a gigabyte; solve LKS, a knapsack whose textbook table would fill four gigabytes, with a single row of eight megabytes; explain why a correct recursive program can crash on a deep input, and rewrite it so that it cannot; and recognize the three quiet ways in which memory also costs time.

6.1 A Suitcase With a Weight Limit

Pak Harun sells batik and woven cloth from Makassar at a weekly market in Surabaya. He flies with a low-cost airline whose ticket includes one cabin bag of 7 kilograms and nothing else; checked luggage costs extra, and a bag found overweight at the gate costs even more. Over the years he has become an expert packer. He does not pack by what an item is worth; he packs by what it *weighs*. He knows that a sarong weighs about 400 grams folded, that a wooden box of the same size weighs three times as much and protects nothing that tissue paper cannot, and that the plastic hangers his supplier ships the shirts on weigh more than the shirts. Before he leaves home he puts the bag on the bathroom scale, because finding out at the gate is too late.

A judge's memory limit is Pak Harun's 7 kilograms. The problem statement says how much memory your program may use, and if it uses more, the judge stops it and reports *Memory limit exceeded*, or, just as often, a runtime error, because the allocation that crossed the line failed. Like the airline, the judge does not care what you meant to pack; it weighs what you actually carried. And like Pak Harun, a good programmer knows what each item weighs: that a `set<int>` holding a million numbers weighs forty times as much as a `vector<int>` holding the same numbers, that a Python list of integers weighs ten times as much as an `array`, and that some structures, like the hangers, cost more than what they hold.

On SPOJ the default memory limit is 1536 MB [1], which sounds generous, and for most problems it is. But the problems in this chapter are exactly those in which the obvious data structure needs several gigabytes, and other judges are stricter: 256 MB is common on many contest platforms. More importantly, memory is not only a limit. Every byte a program touches must travel from the main memory to the processor, and moving bytes takes time. A program that packs lightly is usually also a program that runs fast.

6.2 The Idea in Plain Words

6.2.1 Where a program keeps its data

A running program's memory is divided into regions with different rules. Two of them matter for contest code.

> **DEFINITION.** A *byte* is 8 bits; 1 KB (kibibyte) is 1024 bytes, 1 MB is 1024 KB and 1 GB is 1024 MB. The *stack* is a region of fixed, fairly small size (typically 1 MB on Windows and 8 MB on Linux) that holds the local variables of the functions currently running; each call adds a *frame* and each return removes it. The *heap* is the large region from which containers such as `std::vector` and all Python objects get their memory; it grows on request up to the memory limit. The *peak memory* of a program is the largest amount it held at any one moment; judges measure the peak.

The distinction matters because the two regions fail differently. Asking the heap for more than the limit makes the allocation fail: in C++ `new` throws `std::bad_alloc`, and the program stops with a runtime error unless it catches it. Overflowing the stack gives no warning at all: the program touches memory beyond the end of the stack and is killed with a *segmentation fault*. Section 6.6 shows both.

6.2.2 The weighing formula

Most of the memory of a contest program is in a few large arrays or containers, and each one costs about

$$\text{bytes} \approx n \times b \tag{6.1}$$

where n is the number of elements and b the bytes per element. The formula is trivial; the work is knowing b, which is not always the size of the value you store. A `vector<int>` costs 4 bytes per element, exactly the size of an `int`. A `set<int>` stores the same `int` inside a *node* with three pointers and a color flag, and costs 40. Section 6.3 measures b for the containers you are likely to use.

Two numbers are worth memorizing for the check you should do before coding:

- 10^6 elements of 4 bytes are about 4 MB; 10^8 elements of 4 bytes are about 400 MB;
- 10^9 of anything does not fit.

If Equation (6.1) gives more than half the limit, stop and look for a structure that stores less, because the program also needs memory for input, output, the standard library and the interpreter.

6.2.3 Memory also costs time

Three effects make memory choices matter for the time limit as well.

1. **Allocation.** Creating a node for every element, as `set` and `list` do, calls the memory allocator once per element; creating one array of a million elements calls it once.
2. **Cache.** The processor keeps recently used memory in small, fast *caches*. An array is contiguous, so reading element i brings elements $i + 1$, $i + 2$, … into the cache for free. The nodes of a tree or a linked list are scattered, and each step may wait for main memory, which is roughly a hundred times slower than the fastest cache [2].

3. **Initialization.** Clearing an array of n elements costs n steps. An algorithm that allocates a fresh gigabyte for each test case spends its time clearing, not computing.

The practical rule that follows is the one Pak Harun would give: pack less, and pack it tightly.

6.3 What One Element Really Costs

6.3.1 Measuring it in C++

Guessing the size of a container is unreliable, because it depends on the library's implementation. Measuring is easy. A C++ container gets its memory through an *allocator*, an object with two functions: `allocate(n)`, which returns room for n elements, and `deallocate(p, n)`, which gives it back. **Listing 6.1** defines an allocator `Counting` that does exactly what the standard one does and also keeps a running total of the bytes in use. Each container is built with a million values while the total is watched.

Listing 6.1 `memsize.cpp`: a counting allocator measures the heap bytes that each container uses for 10^6 values.

```
// How many bytes does one element really cost? A counting allocator records
// every heap byte a container asks for while it holds N = 1,000,000 values.
#include <bitset>
#include <cstdio>
#include <functional>
#include <list>
#include <map>
#include <set>
#include <unordered_set>
#include <vector>

static long long liveBytes = 0;             // heap bytes currently in use

template <class T>
struct Counting {                           // a minimal C++11 allocator
    using value_type = T;
    Counting() = default;
    template <class U> Counting(const Counting<U>&) {}
    T* allocate(std::size_t n) {
        liveBytes += (long long)(n * sizeof(T));
        return std::allocator<T>().allocate(n);
    }
    void deallocate(T* p, std::size_t n) {
        liveBytes -= (long long)(n * sizeof(T));
        std::allocator<T>().deallocate(p, n);
    }
};
template <class T, class U>
bool operator==(const Counting<T>&, const Counting<U>&) { return true; }
template <class T, class U>
bool operator!=(const Counting<T>&, const Counting<U>&) { return false; }

const int N = 1000000;

void report(const char* name, long long bytes) {
    std::printf("%-22s %6.2f bytes/element\n", name, (double)bytes / N);
}
```

Listing 6.1 (Continued) *memsize.cpp: a counting allocator measures the heap bytes that each container uses for 10^6 values.*

```
int main() {
    { std::vector<char, Counting<char>> v(N); report("vector<char>", liveBytes); }
    { std::vector<bool, Counting<bool>> v(N); report("vector<bool>", liveBytes); }
    { std::vector<int, Counting<int>> v(N); report("vector<int>", liveBytes); }
    { std::vector<long long, Counting<long long>> v(N);
      report("vector<long long>", liveBytes); }
    { std::vector<int, Counting<int>> v;
      for (int i = 0; i < N; i++) v.push_back(i);
      report("vector<int> push_back", liveBytes); }
    report("bitset<N>", (long long)sizeof(std::bitset<N>));
    { std::list<int, Counting<int>> l;
      for (int i = 0; i < N; i++) l.push_back(i);
      report("list<int>", liveBytes); }
    { std::set<int, std::less<int>, Counting<int>> s;
      for (int i = 0; i < N; i++) s.insert(i);
      report("set<int>", liveBytes); }
    { std::map<int, int, std::less<int>, Counting<std::pair<const int, int>>> m;
      for (int i = 0; i < N; i++) m[i] = i;
      report("map<int,int>", liveBytes); }
    { std::unordered_set<int, std::hash<int>, std::equal_to<int>, Counting<int>> s;
      for (int i = 0; i < N; i++) s.insert(i);
      report("unordered_set<int>", liveBytes); }
    struct Padded { char tag; int value; char flag; };  // 6 bytes of data
    struct Packed { int value; char tag; char flag; };  // same data, reordered
    std::printf("sizeof(Padded) = %zu, sizeof(Packed) = %zu\n",
                sizeof(Padded), sizeof(Packed));
    return 0;
}
```

Each measurement sits in its own pair of braces, so the container is destroyed, and its memory returned, before the next one is built; that is why `liveBytes` can be read directly. The allocator is written in the minimal form that C++11 allows: a `value_type`, a converting constructor, `allocate`, `deallocate` and the two comparison operators. The `std::bitset` cannot take an allocator, so for it we print its `sizeof`, which is its whole size. Compiled with g++ 13.3, the program printed:

```
vector<char>              1.00 bytes/element
vector<bool>              0.12 bytes/element
vector<int>               4.00 bytes/element
vector<long long>         8.00 bytes/element
vector<int> push_back     4.19 bytes/element
bitset<N>                 0.12 bytes/element
list<int>                24.00 bytes/element
set<int>                 40.00 bytes/element
map<int,int>             40.00 bytes/element
unordered_set<int>       27.58 bytes/element
sizeof(Padded) = 12, sizeof(Packed) = 8
```

6.3.2 Reading the numbers

The first five rows are *contiguous* containers, which store the values side by side. They cost what the values cost: 1 byte for a `char`, 4 for an `int`, 8 for a `long long`. The row `vector<int> push_back` is slightly more than 4, because a vector that grows by `push_back` doubles its capacity whenever it is full; after a million pushes the capacity is $2^{20} = 1\,048\,576$, and $1\,048\,576 \times 4 / 10^6 = 4.19$ bytes per element. Calling `reserve(n)` first, or creating the vector with its final size, removes the extra 5%. During each doubling the old and the new block exist at the same time, so the *peak* of a growing vector can be three times the size it ends with; for large arrays, size them once.

`vector<bool>` and `bitset` store one bit per element, eight times less than a `char`. They are the right choice for large tables of yes/no flags, with two cautions. `vector<bool>` is a special case in the standard: its elements are not real `bool` objects, so `&v[i]` is not a `bool*`, and code that takes references to elements, or reads and writes different elements from several threads, behaves differently from every other vector [3]. And reading a bit needs a shift and a mask, so a `vector<char>` is often slightly faster when memory allows it.

The last four rows are *node-based* containers. Each element lives in a separately allocated node:

- a `list<int>` node holds two pointers (next and previous, 8 bytes each) and the `int`, rounded up to 24 bytes;
- a `set<int>` or `map<int,int>` node is a node of a red-black tree: three pointers (parent, left, right), a color, and the value, 40 bytes;
- an `unordered_set<int>` node holds one pointer and the value (16 bytes), plus an array of buckets of 8 bytes each, of which there are about 1.45 for every element at this size.

These numbers are the bytes the containers *ask for*. The system allocator adds its own bookkeeping to every block, typically 8 to 16 bytes, so the true cost of a node is larger still: a `set<int>` of a million elements occupies about 48 MB, twelve times the `vector<int>`. Finally, the last line shows that even a single `struct` can waste space. The `int` in `Padded` must start at an address divisible by 4, so the compiler inserts three empty bytes after the first `char`, and three more at the end so that arrays of the struct stay aligned. Putting the largest field first, as in `Packed`, saves a third.

6.3.3 Measuring it in Python

In Python every value is an *object* with a header, and a list holds pointers to objects, not the values themselves. **Listing 6.2** uses the standard module `tracemalloc`, which counts every byte the interpreter allocates [4].

Listing 6.2 `memsize.py`: `tracemalloc` measures the bytes per element of Python containers holding 10^6 values.

```
# What one element costs in Python: tracemalloc counts every byte that
# CPython allocates while a container of N = 1,000,000 values is built.
import array
import tracemalloc

N = 1000000


def measure(name, build):
    tracemalloc.start()
    obj = build()                       # keep it alive while we look
    used, _peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    del obj
    print("%-28s %6.2f bytes/element" % (name, used / N))


measure("bytearray", lambda: bytearray(N))
measure("array('i')", lambda: array.array("i", [0]) * N)
measure("list of None", lambda: [None] * N)
measure("list of small ints (0..255)", lambda: [i % 256 for i in range(N)])
measure("list of distinct ints", lambda: list(range(10**6, 10**6 + N)))
measure("set of ints", lambda: set(range(10**6, 10**6 + N)))
measure("dict int -> int", lambda: {i: i for i in range(10**6, 10**6 + N)})
```

```
bytearray                     1.00 bytes/element
array('i')                    4.00 bytes/element
list of None                  8.00 bytes/element
list of small ints (0..255)   8.45 bytes/element
list of distinct ints        40.00 bytes/element
set of ints                  65.55 bytes/element
dict int -> int              73.94 bytes/element
```

The list of `None` shows the cost of the list itself: one 8-byte pointer per element. When the elements are distinct integers, each one is also an `int` object of 28 bytes, rounded up to 32 by the allocator, for 40 in all. Small integers from -5 to 256 are shared objects that CPython creates once, which is why the list of small integers costs little more than its pointers. A `set` or a `dict` keeps a hash table with empty slots so that lookups stay fast, and costs 65 to 74 bytes per integer. At the top of the table, `bytearray` and `array` store raw values, 1 and 4 bytes each, exactly like a C++ vector. The lesson for Python is blunt: a list of a million integers weighs 40 MB, and ten million weigh 400 MB. For large tables of small numbers or flags, use `bytearray` or `array`.

Figure 6.1 puts both measurements side by side, on a logarithmic scale.

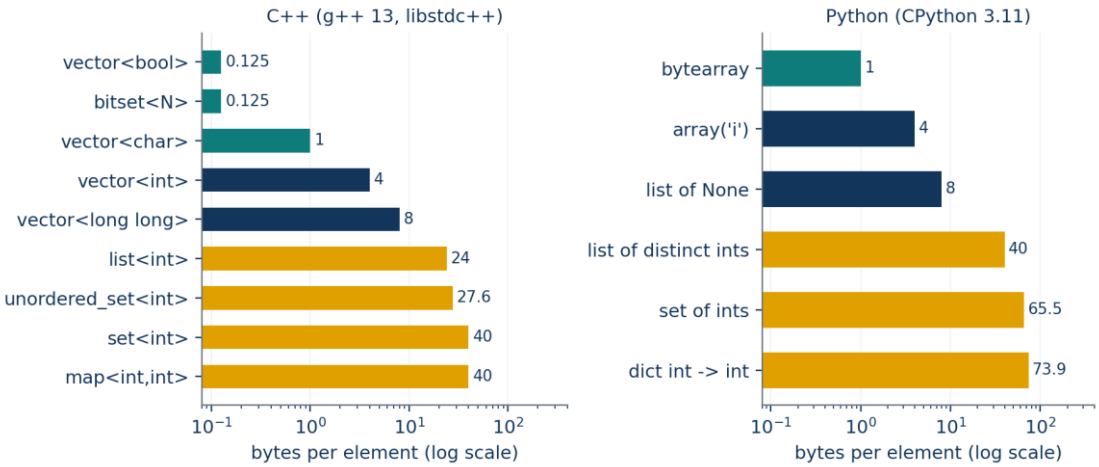


Figure 6.1 Bytes per element for 10^6 values, measured with a counting allocator in C++ and with `tracemalloc` in Python; the scale is logarithmic.

Table 6.1 condenses the measurements into rules you can apply while reading a problem statement.

Table 6.1 Rules of thumb for choosing a container by its weight (g++ 13 with libstdc++, CPython 3.11).

Need	C++ choice	Bytes/element	Python choice	Bytes/element
yes/no flag per number	`vector<char>`, `vector<bool>`	1, 0.125	`bytearray`	1
number below 2^{31}	`vector<int>`	4	`array('i')`	4
number below 2^{63}	`vector<long long>`	8	`array('q')`	8
sorted set with insert/erase	`set<int>`	40 (+ allocator)	sorted list + `bisect`	8 to 40
lookup by key	sort + binary search, or `unordered_map`	8, or 30 to 50	`dict`	about 74

6.4 PRIME1: A Range Too Large to Store

6.4.1 The problem

SPOJ problem PRIME1, "Prime Generator" [5], gives up to 10 test cases, each a pair of numbers m and n with $1 \le m \le n \le 10^9$ and $n - m \le 10^5$. For each pair the program must print every prime p with $m \le p \le n$, one per line, with an empty line between the test cases. The time limit is 6 seconds and the memory limit 1536 MB.

A *prime* is a whole number greater than 1 whose only divisors are 1 and itself: 2, 3, 5, 7, 11, and so on. The classic way to list the primes up to a limit L is the *sieve of Eratosthenes*: write down all numbers from 2 to L, and for each number p that is not yet crossed out, cross out its multiples $p^2, p^2 + p, p^2 + 2p, \ldots$. What remains uncrossed is prime.

6.4.2 Weighing the obvious solution

The obvious solution sieves up to 10^9 once and prints the requested ranges. By Equation (6.1), with one flag per number,

$$10^9 \times 1 \text{ byte} \approx 954 \text{ MB}, \qquad 10^9 \times 4 \text{ bytes} \approx 3.7 \text{ GB} \tag{6.2}$$

as `char` and as `int` respectively. The `int` version is far over the limit; the `char` version squeezes under SPOJ's 1536 MB but would fail on any 256 MB judge, and crossing out a billion flags takes several seconds before the first line is printed. A `vector<bool>` would need 119 MB, which fits, but still spends its time sieving 999,900,000 numbers that nobody asked about. The requested ranges together hold at most $10 \times 100\,001$ numbers; the problem is to sieve only those.

6.4.3 The idea: a small sieve and a window

Two facts make it possible.

Fact 1. Every composite number $x \le n$ has a prime factor $p \le \sqrt{n}$. Indeed, if $x = a \times b$ with $1 < a \le b$, then $a \times a \le a \times b = x \le n$, so $a \le \sqrt{n}$, and any prime factor of a is at most a. Since $\sqrt{10^9} < 31\,623$, the primes up to 31,623 suffice to recognize every composite number up to 10^9. There are only 3401 of them.

Fact 2. To cross out the multiples of p inside the *window* $[m, n]$ we do not need the numbers below m. The first multiple of p that is at least m is

$$f(p) = \left\lceil \frac{m}{p} \right\rceil \cdot p = \left\lfloor \frac{m + p - 1}{p} \right\rfloor \cdot p \tag{6.3}$$

where $\lceil x \rceil$ is x rounded up; the second form uses only integer division. From there, the multiples are $f(p), f(p) + p, f(p) + 2p, \ldots$ up to n. We start at $\max(f(p), p^2)$ instead of $f(p)$, for two reasons: the multiples below p^2 have a smaller prime factor and are crossed out by it anyway, and, more importantly, p itself may lie inside the window and must not be crossed out.

So the segmented sieve works with two small arrays: a sieve of 31,624 flags that finds the 3401 small primes once, and a window of $n - m + 1 \le 100\,001$ flags, one per number in the current test case, reused for every test. **Figure 6.2** shows the picture.

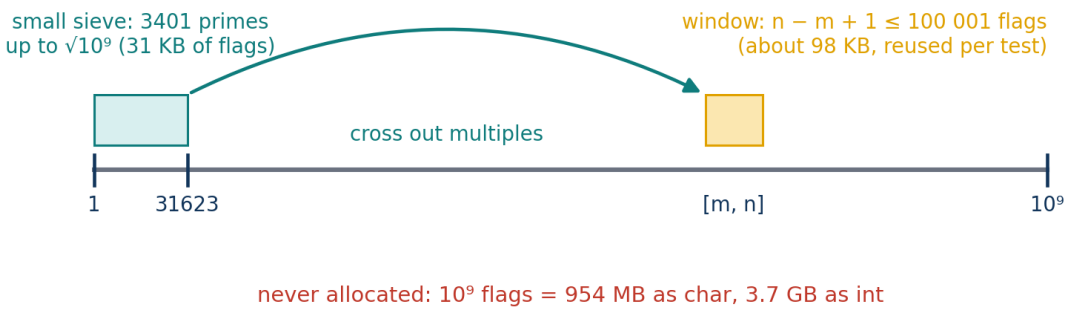


Figure 6.2 The segmented sieve: the small primes up to $\sqrt{10^9}$ cross out their multiples inside the window $[m, n]$; the numbers outside the window are never stored.

How much work is it? For each small prime p the window receives about $(n - m)/p$ crossings, so the total is

$$\sum_{p \le \sqrt{n}} \frac{n-m}{p} + \pi\left(\sqrt{n}\right) \approx (n-m)\ln\ln\sqrt{n} + \pi\left(\sqrt{n}\right) \tag{6.4}$$

using the classical estimate that the sum of $1/p$ over the primes up to x grows like $\ln\ln x$ [6], and where $\pi(x)$ is the number of primes up to x (the second term counts the primes whose loop does nothing). With $n - m = 10^5$ and $\ln\ln 31\,623 \approx 2.3$, that is about 230,000 crossings and 3401 loop starts per test case: tiny. Memory: $31\,624 + 100\,001$ bytes of flags and 3401×4 bytes of primes, about 140 KB in all, against the 954 MB of Equation (6.2).

6.4.4 A worked example by hand

Take the window [24,30]. The window has 7 numbers, and $\sqrt{30} < 6$, so only the primes 2, 3 and 5 matter.

- $p = 2$: by Equation (6.3), $f(2) = \lfloor 25/2 \rfloor \times 2 = 24$, and $\max(24,4) = 24$. Cross out 24, 26, 28, 30.
- $p = 3$: $f(3) = \lfloor 26/3 \rfloor \times 3 = 24$. Cross out 24, 27, 30.
- $p = 5$: $f(5) = \lfloor 28/5 \rfloor \times 5 = 25$. Cross out 25, 30.
- $p = 7$: $7^2 = 49 > 30$, stop.

The uncrossed numbers are 29 only, and 29 is indeed the only prime between 24 and 30.

6.4.5 The programs

Listing 6.3 is the C++ program.

Listing 6.3 `prime1.cpp`: the segmented sieve; memory depends on $\sqrt{n}$ and on $n - m$, never on n.

```cpp
// PRIME1 (SPOJ): all primes in [m, n], n <= 10^9, n - m <= 10^5.
// Segmented sieve: memory depends on sqrt(n) and on n - m, never on n.
#include <cstdio>
#include <vector>

int main() {
    const int LIMIT = 31623;                 // 31623^2 > 10^9
    std::vector<char> composite(LIMIT + 1, 0);
    std::vector<int> small;                  // primes up to sqrt(10^9)
    for (int i = 2; i <= LIMIT; i++) {
        if (composite[i]) continue;
        small.push_back(i);
        for (long long j = (long long)i * i; j <= LIMIT; j += i) composite[j] = 1;
    }
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    for (int tc = 0; tc < t; tc++) {
        long long m, n;
        if (std::scanf("%lld %lld", &m, &n) != 2) break;
        if (tc > 0) std::printf("\n");       // blank line between test cases
        std::vector<char> crossed(n - m + 1, 0);  // crossed[k] is about m + k
        for (int p : small) {
            long long pp = (long long)p * p;
            if (pp > n) break;
            long long first = (m + p - 1) / p * p;   // first multiple >= m
            if (first < pp) first = pp;              // smaller ones: done by others
            for (long long x = first; x <= n; x += p) crossed[x - m] = 1;
        }
        for (long long x = m; x <= n; x++)
            if (x >= 2 && !crossed[x - m]) std::printf("%lld\n", x);
    }
    return 0;
}
```

Three details are worth a second look. The products `(long long)i * i` and `(long long)p * p` are computed in 64 bits: $31\,623^2$ fits in an `int`, but the habit costs nothing and the next problem may not be so kind. The window is a local `vector<char>` created for each test case, so it is always cleared; a global array would need clearing by hand, and forgetting to clear it is the classic bug of Chapter 7. And the number 1 is excluded by the test `x >= 2`, because nothing crosses it out: it has no prime factor at all.

Listing 6.4 is the Python program. Instead of a loop that crosses out one number at a time, it assigns a whole slice: `alive[first - m::p] = bytes(k)` sets every p-th flag from `first` on to 0 in a single operation that runs at C speed inside the interpreter. The length `k` of the zero-filled `bytes` object must match the number of positions in the slice exactly, which `len(range(first, n + 1, p))` computes without building the range.

Listing 6.4 `prime1.py`: the segmented sieve on `bytearray` windows, crossing out whole slices at once.

```
# PRIME1 (SPOJ): all primes in [m, n] with a segmented sieve on bytearrays.
# Slice assignment crosses out a whole arithmetic progression at C speed.
import sys


def small_primes(limit):
    flags = bytearray([1]) * (limit + 1)
    flags[0] = flags[1] = 0
    for i in range(2, int(limit ** 0.5) + 1):
        if flags[i]:
            flags[i * i::i] = bytes(len(range(i * i, limit + 1, i)))
    return [i for i in range(limit + 1) if flags[i]]


def main():
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    small = small_primes(31623)
    blocks = []
    for tc in range(t):
        m, n = int(data[1 + 2 * tc]), int(data[2 + 2 * tc])
        alive = bytearray([1]) * (n - m + 1)     # alive[k] is about m + k
        for p in small:
            if p * p > n:
                break
            first = max(p * p, (m + p - 1) // p * p)
            if first <= n:
                alive[first - m::p] = bytes(len(range(first, n + 1, p)))
        for x in range(m, min(2, n + 1)):        # 1 is not a prime
            alive[x - m] = 0
        blocks.append("\n".join(str(m + k) for k in range(n - m + 1) if alive[k]))
    sys.stdout.write("\n\n".join(b for b in blocks) + "\n")


if __name__ == "__main__":
    main()
```

We test with three windows of our own: one that contains the smallest primes and the number 1, one at the very top of the range, and the one we worked by hand.

```
3
1 20
999999937 1000000000
24 30
```

```
2
3
5
7
11
13
17
19

999999937

29
```

The Python program prints the same lines. The number 999999937 is the largest prime below 10^9, and the windows are separated by empty lines as the problem asks. On the heaviest input the problem allows, ten windows of 100,001 numbers each at the top of the range, the C++ program took 0.01 seconds and the Python program 0.05 seconds, and both stayed near 10 MB, most of which is the program and the interpreter themselves.

6.5 LKS: When the Table Does Not Fit

6.5.1 The problem

SPOJ problem LKS, "Large Knapsack" [7], is the classic 0/1 knapsack problem with large numbers. A knapsack has capacity K, and there are N items, item i with value v_i and weight w_i. Choose a set of items, each at most once, whose total weight is at most K and whose total value is as large as possible; print that value. The limits are $K \le 2\,000\,000$ and $N \le 500$, values and weights up to 10^7, and the problem guarantees that every intermediate result fits in a signed 32-bit integer. The time limit is 2 seconds.

Pak Harun's suitcase is this problem: $K = 7000$ grams, the items are the pieces of cloth he could take, and the values are what he expects to sell them for.

6.5.2 The textbook solution and its weight

Chapter 9 develops dynamic programming in depth; here we need only the standard recurrence. Let $B_i(c)$ be the best value using only the first i items with capacity c. Either item i is left out, and the best is $B_{i-1}(c)$, or it is taken, which needs $c \ge w_i$ and gives $B_{i-1}(c - w_i) + v_i$. So

$$B_i(c) = \max(B_{i-1}(c),\ B_{i-1}(c - w_i) + v_i), \qquad B_0(c) = 0 \tag{6.5}$$

where the second option exists only when $c \ge w_i$, and the answer is $B_N(K)$ [8]. The textbook stores the whole table, one row per item and one column per capacity. **Listing 6.5** is that program.

Listing 6.5 `lks_table.cpp`: the knapsack with the full table; correct, but the largest tests need about 4 GB.

```cpp
// LKS with the textbook full table: table[i][c] for every item i and every
// capacity c. Correct, and fine for small K -- but the largest tests need
// (N + 1) * (K + 1) ints, about 4 GB, far above the judge's 1536 MB.
#include <cstdio>
#include <vector>

int main() {
    int K, N;
    if (std::scanf("%d %d", &K, &N) != 2) return 0;
    std::vector<std::vector<int>> table(N + 1, std::vector<int>(K + 1, 0));
    for (int i = 1; i <= N; i++) {
        int value, weight;
        if (std::scanf("%d %d", &value, &weight) != 2) return 0;
        for (int c = 0; c <= K; c++) {
            table[i][c] = table[i - 1][c];                    // skip item i
            if (c >= weight && table[i - 1][c - weight] + value > table[i][c])
                table[i][c] = table[i - 1][c - weight] + value;  // take it
        }
    }
    std::printf("%d\n", table[N][K]);
    return 0;
}
```

Weigh it with Equation (6.1): the table has $(N + 1)(K + 1)$ entries of 4 bytes, so at the limits it needs

$$501 \times 2\,000\,001 \times 4 \text{ bytes} = 4\,008\,002\,004 \text{ bytes} \approx 3.73 \text{ GB} \tag{6.6}$$

two and a half times SPOJ's 1536 MB. On a small input the program is perfectly correct. The file below is the example worked by hand in Section 6.5.4: capacity 10 and four items, each line giving a value and a weight.

```
10 4
6 5
5 4
4 3
7 6
```

```
12
```

Section 6.7 checks it against a brute force on hundreds of random inputs. To see what happens on the judge at full size, we imitated its limit on Linux with `ulimit -v 1572864`, which caps the program's memory at 1536 MB, and ran the largest test that `gen_lks_max.py` generates ($K = 2\,000\,000$, $N = 500$):

```
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted
```

The allocation of the table failed, `std::bad_alloc` was thrown, nobody caught it, and the program aborted: on the judge, a runtime error or *Memory limit exceeded*, depending on where the limit is enforced.

6.5.3 One row is enough

Look at Equation (6.5) again: row i is computed from row $i-1$ only. Rows 0,1, … , $i-2$ are never read again. So we can keep a single row `best` and overwrite it in place, as long as, when we compute the new `best[c]`, the value `best[c - w]` we read is still the *old* one, from row $i-1$.

That depends on the order of the loop. The new value at c reads the old value at the smaller index $c-w$. If we go through the capacities from high to low, K down to w, then at the moment we write `best[c]` the entry `best[c - w]` has not yet been overwritten in this pass, because $c-w < c$ comes later. It still holds row $i-1$, and Equation (6.5) is computed exactly. If we went from low to high, `best[c - w]` would already hold row i, which might include item i, and the item could be taken twice, three times, as many times as it fits: that computes a different problem, the *unbounded* knapsack. Chapter 9 revisits this direction trick; here its value is memory. **Listing 6.6** is the program.

Listing 6.6 `lks.cpp`: the knapsack with a single row of $K+1$ values, visited from high to low capacity.

```cpp
// LKS (SPOJ): 0/1 knapsack with capacity K <= 2,000,000 and N <= 500 items.
// A full table would need (N + 1) * (K + 1) ints, about 4 GB. One row of
// K + 1 ints (8 MB) is enough if capacities are visited from high to low.
#include <cstdio>
#include <vector>

int main() {
    int K, N;
    if (std::scanf("%d %d", &K, &N) != 2) return 0;
    std::vector<int> best(K + 1, 0);   // best[c]: max value with weight <= c
    for (int i = 0; i < N; i++) {
        int value, weight;
        if (std::scanf("%d %d", &value, &weight) != 2) return 0;
        for (int c = K; c >= weight; c--) {      // high to low: item used once
            int take = best[c - weight] + value;
            if (take > best[c]) best[c] = take;
        }
    }
    std::printf("%d\n", best[K]);
    return 0;
}
```

Its memory is $(K + 1) \times 4$ bytes, 8 MB at the limit, 500 times less than Equation (6.6), and its time is the same $N \times K = 10^9$ simple steps. On the largest test it took 0.48 seconds on our machine (the best of three runs; up to 1.2 seconds while other jobs shared the processor), inside the 2-second limit; its inner loop runs through memory in order, which the cache rewards.

6.5.4 A worked example by hand

Take capacity $K = 10$ and four items with (value, weight) = (6, 5), (5, 4), (4, 3) and (7, 6). **Table 6.2** shows the row after each item.

Table 6.2 The single row `best` of `lks.cpp` after each item, for capacity 10.

After item	$c=0$	1	2	3	4	5	6	7	8	9	10
(start)	0	0	0	0	0	0	0	0	0	0	0
(6, 5)	0	0	0	0	0	6	6	6	6	6	6
(5, 4)	0	0	0	0	5	6	6	6	6	11	11
(4, 3)	0	0	0	4	5	6	6	9	10	11	11
(7, 6)	0	0	0	4	5	6	7	9	10	11	12

Follow one entry. When item (5, 4) arrives, the loop reaches $c = 9$ first and computes $\max(6, \text{best}[5] + 5) = \max(6, 6 + 5) = 11$: item (6, 5) and item (5, 4) together. Because we go downward, `best[5]` still holds 6, the value *before* item (5, 4). Had we gone upward, `best[5]` would already have been updated, and `best[9]` might have been built from item (5, 4) twice. The final answer is best[10] = 12, items (5, 4) and (7, 6).

The same input file gives the same answer with the one-row program, as it did with the full table:

```
10 4
6 5
5 4
4 3
7 6
```

```
12
```

6.5.5 Python cannot pass LKS

Listing 6.7 is the same one-row algorithm in Python. To avoid a Python loop over the capacities, it builds the shifted row with a slice and combines the two rows with `map(max, ...)`, which runs mostly inside the interpreter's C code. Because the right-hand side is computed completely before the assignment, it reads only old values, and the direction problem of Section 6.5.3 disappears.

Listing 6.7 `lks.py`: the one-row knapsack in Python; correct, but far too slow for the largest tests.

```
# LKS (SPOJ) in Python: the same one-row DP. Correct, but N * K can reach
# 10^9 steps; even with list slicing this is too slow for the largest tests.
import sys


def main():
    data = sys.stdin.buffer.read().split()
    k, n = int(data[0]), int(data[1])
    best = [0] * (k + 1)
    for i in range(n):
        value, weight = int(data[2 + 2 * i]), int(data[3 + 2 * i])
        if weight > k:
            continue
        # new[c] = max(best[c], best[c - weight] + value), for all c >= weight,
        # computed from the OLD row, so every item is used at most once
        shifted = [b + value for b in best[:k + 1 - weight]]
        best[weight:] = map(max, best[weight:], shifted)
    print(best[k])


if __name__ == "__main__":
    main()
```

It prints 12 on the sample and agrees with the brute force on every random test. But it is not fast enough. On a test one tenth of the maximum ($K = 200\,000$, $N = 500$) it took 8.8 seconds, against 0.05 seconds for the C++ program; the full test, ten times larger, would need well over a minute. $N \times K = 10^9$ elementary steps is simply beyond an interpreter in 2 seconds, whatever tricks we use, and this is one of the problems in this book that must be solved in C++. The lesson generalizes Chapter 3's budget table: in Python, count on about 10^7 simple operations per second, fewer when each one creates objects.

6.6 The Stack Is Smaller Than You Think

6.6.1 A graph shaped like a line

Recursion is the natural way to write depth-first search (DFS), one of the basic graph explorations (Chapter 12 treats it fully): to visit a vertex, mark it and visit each unmarked neighbor. Each recursive call adds a frame to the stack, holding its local variables and the address to return to, and the frame stays there until the call returns. For a graph shaped like a line, $1 - 2 - 3 - \cdots - n$, the recursion goes n calls deep before the first one returns. **Listing 6.8** builds such a graph and explores it in two ways: recursively, and with an explicit stack of vertices kept in a `vector` on the heap.

Listing 6.8 `deep_dfs.cpp`: depth-first search on a path of n vertices, recursive and with an explicit stack.

```
// Depth-first search on a path 1 - 2 - ... - n, written two ways.
// Recursion keeps one stack frame per level; the explicit stack keeps one
// int per level on the heap. Usage: deep_dfs n recursive|iterative
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <vector>

std::vector<std::vector<int>> adj;
std::vector<char> seen;

int dfsRecursive(int u) {                   // returns how many nodes it saw
    seen[u] = 1;
    int count = 1;
    for (int v : adj[u])
        if (!seen[v]) count += dfsRecursive(v);
    return count;
}

int dfsIterative(int start) {
    std::vector<int> stack{start};          // lives on the heap, grows freely
    seen[start] = 1;
    int count = 0;
    while (!stack.empty()) {
        int u = stack.back();
        stack.pop_back();
        count++;
        for (int v : adj[u])
            if (!seen[v]) { seen[v] = 1; stack.push_back(v); }
    }
    return count;
}

int main(int argc, char** argv) {
    int n = argc > 1 ? std::atoi(argv[1]) : 1000000;
    bool recursive = argc > 2 && std::strcmp(argv[2], "recursive") == 0;
    adj.assign(n + 1, {});
    seen.assign(n + 1, 0);
    for (int u = 1; u < n; u++) {           // edges u - (u + 1)
        adj[u].push_back(u + 1);
        adj[u + 1].push_back(u);
    }
    int visited = recursive ? dfsRecursive(1) : dfsIterative(1);
    std::printf("%s DFS visited %d nodes\n",
                recursive ? "recursive" : "iterative", visited);
    return 0;
```

Listing 6.8 (Continued) *deep_dfs.cpp: depth-first search on a path of n vertices, recursive and with an explicit stack.*

```
}
```

Run without arguments, the program builds a path of a million vertices and uses the iterative version:

```
iterative DFS visited 1000000 nodes
```

6.6.2 Where the recursion breaks

We ran both versions under three stack sizes, set with the Linux command `ulimit -s` (in kilobytes). **Table 6.3** shows the results.

Table 6.3 Recursive and iterative DFS on a path, under different stack sizes (g++ 13.3, `-O2`).

n	Stack	Recursive	Iterative
100000	8 MB (Linux default)	visited 100000 nodes	visited 100000 nodes
1000000	8 MB (Linux default)	Segmentation fault	visited 1000000 nodes
100000	1 MB (Windows default)	Segmentation fault	visited 100000 nodes

The arithmetic explains it. Each frame of `dfsRecursive` holds the parameter, the counter, the loop state, some saved registers and the return address. Increasing n step by step, we found that the recursion still worked at 160,000 levels and crashed at 180,000 with an 8 MB stack, so a frame takes about $8\,388\,608/170\,000 \approx 49$ bytes. A million frames need about 49 MB, six times the stack, while a hundred thousand need about 5 MB, which fits in 8 MB but not in the 1 MB that Windows gives a program's main thread by default [9]. The iterative version keeps one `int` per waiting vertex in a vector on the heap, so its depth is limited only by the memory limit.

What does a judge do? Some judges run programs with a larger stack than a desktop default, and some set the stack as large as the memory limit; but few document it, and a program that relies on it is not portable. When the depth can reach 10^5 or more, write the iteration.

At home you can enlarge the stack to reproduce a judge's behavior or to run a recursive reference solution: on Linux and macOS with `ulimit -s unlimited` (or a size in kilobytes) in the terminal before running the program, and on Windows with MinGW-w64 by linking with `-Wl,--stack,268435456`, which asks for a 256 MB stack. Both change only your own machine; neither is available on the judge, which is exactly why the explicit stack is the portable answer.

6.6.3 The same story in Python

Python protects itself earlier. Its interpreter counts the depth of Python calls and raises `RecursionError` when it exceeds the recursion limit, 1000 by default. The Python version of the program, `deep_dfs.py` in the chapter's folder, stops on a path of 100,000 vertices with

```
RecursionError: maximum recursion depth exceeded
```

`sys.setrecursionlimit(10**6)` raises the counter but not the real stack of the C code underneath, so a deep enough recursion then crashes the whole interpreter instead of raising an exception. The robust answer is the same as in C++: an explicit stack. The iterative Python version handled a million vertices in 0.58 seconds and 170 MB, against 0.09 seconds and 58 MB for C++; the difference in memory is the 40 bytes per integer of **Table 6.1**, paid several times over in the adjacency lists.

That last number points to one more weight. Even the C++ program used 58 MB for a graph with two million adjacency entries, because a `vector<vector<int>>` gives every vertex its own small vector: 24 bytes for the vector object and a separate heap block for its elements. For graphs with millions of vertices, the compact alternative stores the neighbors of all vertices one after another in a single array, together with a second array that records where the list of each vertex starts: two allocations instead of a million.

6.7 Testing It

Every program in this chapter was checked the way Chapter 8 describes: a generator writes small random inputs, a brute force that is obviously right computes the expected output, and the stress harness compares.

- For PRIME1 the brute force tests each number of the range by trial division, dividing by every d with $d \times d \le x$. The generator makes windows of up to 300 numbers, sometimes starting at 1, 2 or 3, sometimes anywhere up to 10^9.
- For LKS the brute force tries all 2^N subsets of at most 12 items, which needs no recurrence and no direction trick at all. **Listing 6.9** is that program.

Listing 6.9 `brute_lks.py`: the LKS brute force tries every subset of the items.

```
# Brute force for LKS: try all 2^N subsets of items.
import sys

data = sys.stdin.read().split()
k, n = int(data[0]), int(data[1])
items = [(int(data[2 + 2 * i]), int(data[3 + 2 * i])) for i in range(n)]
best = 0
for mask in range(1 << n):
    value = weight = 0
    for i in range(n):
        if mask >> i & 1:
            value += items[i][0]
            weight += items[i][1]
    if weight <= k:
        best = max(best, value)
print(best)
```

The four stress runs, with 300 random tests each, gave:

```
python3 stress.py "python3 gen_prime1.py" "python3 brute_prime1.py" build/prime1
python3 stress.py "python3 gen_prime1.py" "python3 brute_prime1.py" \
    "python3 ../python/prime1.py"
python3 stress.py "python3 gen_lks.py" "python3 brute_lks.py" build/lks
python3 stress.py "python3 gen_lks.py" "python3 brute_lks.py" \
    "python3 ../python/lks.py"
```

```
OK: 300 random tests agree
OK: 300 random tests agree
OK: 300 random tests agree
OK: 300 random tests agree
```

`lks_table.cpp` also agreed with the brute force on 200 random tests, which confirms what Section 6.5 claimed: the full table is right and only too heavy. The largest inputs were timed separately with `measure.py`, which reports the wall time and the peak memory of a run; the numbers quoted in Sections 6.4 and 6.5 come from it.

6.8 Using It for Real

The habit this chapter teaches is Pak Harun's bathroom scale: weigh before you travel. In practice it is a three-line worksheet that you fill in while reading the constraints, before writing any code.

1. **List the big structures** your idea needs, with their sizes in terms of the input: "a flag per number up to n", "a table of $N \times K$", "a list of neighbors per vertex".
2. **Weigh each one** with Equation (6.1) and the bytes per element of **Table 6.1**, at the largest input.
3. **Compare with the limit**, keeping half of it in reserve. If a structure does not fit, ask the two questions of this chapter: can I store only a *window* of it (PRIME1)? can I keep only the *last row* (LKS)?

The same worksheet serves outside contests. A cooperative of durian and fruit farmers in Jember that wants to keep one record per tree per day for five years, 12,000 trees, has $12\,000 \times 365 \times 5 \approx 2.2 \times 10^7$ records. As a Python list of dictionaries, at a few hundred bytes each, that is several

gigabytes; as rows of a few numbers in an `array` or a database table, it is a few hundred megabytes; as a daily total per tree that is updated in place, like the single row of LKS, it is a few kilobytes. Which one is right depends on the questions the farmers want to ask, but the weights should be known before the choice is made.

6.9 Common Mistakes, and How to Avoid Them

- **Sizing an array by the value range instead of the input size.** "$n \le 10^9$" does not mean "an array of 10^9". Look for a window, a compression of the values, or a formula.
- **Keeping a table that is read only one row back.** If row i depends only on row $i-1$, keep one or two rows; check the loop direction when you keep one.
- **Using node containers for bulk data.** A `set<int>` or `map<int,int>` of a million elements weighs about ten times a sorted `vector`. When the data does not change after it is read, sort a vector and use binary search.
- **Growing a large vector by `push_back`.** The doubling can make the peak three times the size the vector ends with. Create it with its size, or call `reserve`.
- **Python lists of numbers.** A list of 10^7 integers weighs 400 MB. Use `bytearray` or `array`.
- **Deep recursion.** A recursive DFS on 10^6 vertices crashes at 8 MB of stack, and at 10^5 on Windows. Write an explicit stack.
- **Big local arrays.** `int a[1000000];` inside a function is 4 MB on the stack: fine on Linux, a crash on Windows. Make it a `vector` or a global.
- **Allocating per test case without need.** A fresh gigabyte per test case is cleared every time. Allocate once for the largest case, or allocate exactly what each case needs, as `prime1.cpp` does.

6.10 Chapter Summary

- A judge weighs the peak memory of your program; the stack is small (1 to 8 MB), the heap is large up to the limit, and each fails differently.
- The weight of a structure is $n \times b$ (Equation (6.1)), and b must be measured: 1, 4 or 8 bytes in contiguous containers, 24 to 40 in node containers, 40 to 74 in Python lists, sets and dictionaries of integers (**Figure 6.1**).
- PRIME1's obvious sieve weighs 954 MB (Equation (6.2)); the segmented sieve keeps 3401 small primes and a window of 100,001 flags, about 140 KB, and runs in about $(n-m)\ln\ln\sqrt{n}$ steps (Equation (6.4)).
- LKS's table weighs 3.73 GB (Equation (6.6)); because each row of Equation (6.5) reads only the row before, one row of 8 MB suffices, traversed from high to low capacity. Python is too slow for its 10^9 steps.
- Recursion depth is memory too: a million nested calls overflow an 8 MB stack. An explicit stack on the heap removes the limit, in C++ and in Python.
- Lighter structures are usually faster, because of fewer allocations, better use of the cache and less clearing.

6.11 Exercises

1. A problem gives $n \le 5 \times 10^6$ integers below 10^9 and asks how many distinct values there are. Weigh three solutions in C++: a `set<int>`, an `unordered_set<int>`, and a sorted `vector<int>` followed by counting changes. Which would you submit on a 64 MB judge?

2. For the window [1,12], run the segmented sieve by hand as in Section 6.4.4. Which line of `prime1.cpp` prevents the small primes 2, 3, 5, 7 and 11 from being crossed out?
3. Change `prime1.cpp` so that the window is a `vector<bool>`. Does the output change? Measure the time on the largest input with `measure.py` and explain the result.
4. In `lks.cpp`, reverse the inner loop to run from `weight` up to `K`. What does the program print for the sample of Section 6.5.4, and which problem does it now solve?
5. The largest LKS test has $K = 2 \times 10^6$ but, in our generator, the total weight of all items is only about 10^6. Improve `lks.cpp` so that the inner loop never visits a capacity larger than the total weight of the items read so far, and measure the gain.
6. Write the recursive DFS of `deep_dfs.py` so that it can handle $n = 10^6$ in Python without an explicit stack, using `sys.setrecursionlimit` and `threading.stack_size`. Why is the explicit stack still the better answer?

6.12 Solutions to the Exercises

1. With **Table 6.1**: the `set<int>` needs about $5 \times 10^6 \times 40 = 200$ MB plus allocator overhead, the `unordered_set<int>` about $5 \times 10^6 \times 28 = 140$ MB, and the sorted vector $5 \times 10^6 \times 4 = 20$ MB. Only the vector fits in 64 MB. Sorting costs $O(n \log n)$, about 10^8 basic steps, well under a second, and after sorting the distinct values are counted by one pass that compares each element with the one before (or with `std::unique`).

2. $\sqrt{12} < 4$, so only $p = 2$ and $p = 3$ cross anything. For $p = 2$: $f(2) = \lfloor 2/2 \rfloor \times 2 = 2$, but the start is $\max(2,4) = 4$: cross 4, 6, 8, 10, 12. For $p = 3$: $f(3) = 3$, start $\max(3,9) = 9$: cross 9, 12. For $p = 5$, $25 > 12$: stop. The uncrossed numbers from 2 on are 2, 3, 5, 7, 11, and 1 is excluded by `x >= 2`. The line `if (first < pp) first = pp;` protects the small primes: without it, $p = 2$ would start at $f(2) = 2$ and cross out 2 itself.

3. The output does not change: only the storage of the flags does. We measured 0.010 to 0.015 seconds for the `vector<bool>` version against 0.007 to 0.009 for the `vector<char>` version: slightly worse, because each access to a `vector<bool>` element needs a shift and a mask, and with a window of only 100,001 flags the memory saving (86 KB) is irrelevant. `vector<bool>` pays off when the byte version would not fit, not otherwise.

4. It prints 13. With an upward loop, `best[c - weight]` has already been updated for the current item, so an item can be counted several times: the program now solves the *unbounded* knapsack, in which each item may be taken any number of times. For the sample the best unbounded choice is item (4, 3) twice and item (5, 4) once, weight 10 and value 13. Reverse the loop back to recover the 0/1 answer 12.

5. Keep a running sum `total` of the weights read so far and loop `for (int c = std::min(K, total); c >= weight; c--)`, adding `weight` to `total` before the loop. Capacities above the total weight cannot hold anything better than the total weight itself, and `best[c]` for those c equals `best[total]`; so at the end the answer is `best[std::min(K, total)]`. On our largest test, where all 500 weights together are 951,080, the loop never runs above that total, and while the first items are read it runs much lower still; the time dropped from 0.48 to 0.16 seconds. The program must print `best[min(K, total)]`, not `best[K]`, because the high entries are never filled.

6. Run the search in a new thread with a large stack:

```
import sys, threading
sys.setrecursionlimit(1 << 21)
threading.stack_size(512 * 1024 * 1024)
t = threading.Thread(target=main)
t.start()
t.join()
```

This works on Linux and usually on Windows, but it depends on how much stack the operating system lets a thread have, it uses much more memory than the explicit stack, and on some judges thread creation is restricted. The explicit stack depends on nothing: it is ordinary heap memory, bounded only by the memory limit.

References

[1] Wikipedia, “SPOJ.” [Online]. Available: https://en.wikipedia.org/wiki/SPOJ (accessed Sep. 2026).

[2] U. Drepper, “What Every Programmer Should Know About Memory,” Red Hat, Inc., Nov. 2007. [Online]. Available: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

[3] cppreference.com, “std::vector.” [Online]. Available: https://en.cppreference.com/w/cpp/container/vector_bool (accessed Sep. 2026).

[4] Python Software Foundation, “tracemalloc: Trace memory allocations,” *The Python Standard Library*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/tracemalloc.html

[5] Sphere Online Judge, “PRIME1 - Prime Generator.” [Online]. Available: https://www.spoj.com/problems/PRIME1/ (accessed Sep. 2026).

[6] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, 6th ed., revised by D. R. Heath-Brown and J. H. Silverman. Oxford, UK: Oxford University Press, 2008.

[7] Sphere Online Judge, “LKS - Large Knapsack.” [Online]. Available: https://www.spoj.com/problems/LKS/ (accessed Sep. 2026).

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

[9] Microsoft, “/STACK (Stack allocations),” *Linker options*, Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/cpp/build/reference/stack-stack-allocations (accessed Sep. 2026).

Chapter 7

Classic Implementation Gotchas That Cause Wrong Answer

The most frustrating verdict on a judge is *Wrong answer* for a program whose idea is right. The algorithm is correct, the proof is sound, the samples pass, and still the judge says no. This chapter is a field guide to the small implementation slips that cause most of those verdicts. By the end of it you will be able to recognize the seven families of such slips and the symptom each one leaves; solve four real SPOJ problems, EIGHTS, FCTRL2, ADDREV and CANDY, each chosen because it contains one classic trap, and prove each solution correct; predict the results of integer division, remainders and floating-point arithmetic in C++ and in Python, which differ; and catch every slip of the chapter with the same stress test that Chapter 8 develops in full.

7.1 Ten Meters of Fence, Eleven Posts

Mas Dedi is a carpenter in a village outside Klaten. A neighbor asks him to put up a bamboo fence ten meters long along the front of her yard, with a post every meter. Mas Dedi has done this a hundred times, and still, the first time he did it, he bought ten posts. Ten meters, a post per meter, ten posts: the reasoning sounds perfect. At the end of the afternoon he stood at the last meter with no post in his hand, because a fence of ten sections has *eleven* posts, one at each end of every section. Carpenters call this the fence-post error, and programmers have borrowed the name for the loop that runs one time too few or one time too many.

His cousin Mbak Wati keeps the till at a family grocery. Her old calculator shows eight digits. One evening, adding up a month of deliveries, she typed a total of ninety million rupiah plus fifteen million, and the display showed an “E” and a meaningless number. The calculator had not made a mistake in arithmetic; the answer simply did not fit in its eight digits. A C++ `int` is Mbak Wati’s calculator with a slightly bigger display: about two thousand million, and then garbage. Unlike the calculator, it does not even show the “E”.

Neither Mas Dedi nor Mbak Wati misunderstood the problem. Their *method* was right; a detail of carrying it out was wrong. That is exactly the situation of a contestant whose correct idea gets *Wrong answer*, and the cure is also the same: know the handful of places where such details hide, and check them on purpose. A carpenter who has been burned once counts the posts as sections plus one; a shopkeeper who has seen the “E” adds large sums in two halves. This chapter gives you the programmer’s version of those habits.

7.2 The Idea in Plain Words

> **DEFINITION.** An *implementation gotcha* is a slip in turning a correct idea into code that makes the program give a wrong output on some valid inputs while it still looks right on the others. It is not a wrong algorithm: fixing it changes a line or two, not the approach. The *symptom* of a gotcha is the pattern of inputs on which it fails.

Symptoms matter, because the judge tells you only that *some* test failed. Knowing which families of slips produce which symptoms lets you guess where to look. **Table 7.1** lists the seven families this chapter covers, with the symptom that usually betrays each one and the section that treats it.

Table 7.1 Seven families of implementation gotchas, the symptom each leaves, and where this chapter treats it.

Family	Typical symptom	Section
Integer overflow	right on small inputs, wrong (often negative) on large ones	7.3
Numbers beyond every built-in type	right up to a threshold, then wrong or zero	7.4
Format of numbers: leading and trailing zeros	right except when a result starts or ends with 0	7.5
State that leaks between test cases	first test case right, later ones wrong	7.6
Division, remainder, rounding	wrong only for negative numbers or exact halves	7.7
Floating-point comparison and printing	wrong in the last digit, or on "exact" values	7.7
Off-by-one and input shape	wrong at the ends of ranges, or on unusual spacing	7.8

Every family below comes with a real SPOJ problem or a small demonstration, a correct program in C++ and Python, and, where it helps, the broken program that a careful person could easily write. We keep the broken programs deliberately: seeing the exact wrong output is how the symptom becomes recognizable.

7.3 Overflow: EIGHTS

7.3.1 The problem

SPOJ problem EIGHTS, "Triple Fat Ladies" [1], asks for the k-th smallest positive integer whose cube ends in the digits 888. The input gives the number of queries t and then t values of k, with $1 \le k \le 2 \times 10^{12}$; the answer is guaranteed to be below 2^{63}. The time limit is about 1.2 seconds.

Searching upward and testing cubes is hopeless for k near 2×10^{12}, so we first understand the numbers.

7.3.2 The idea, with a proof

Let us find all n below 1000 whose cube ends in 888, by brute force: they are 192, 442, 692 and 942. Check the first: $192^3 = 7\,077\,888$. The four numbers are 250 apart, which suggests that the answers form an *arithmetic sequence*: start at 192 and add 250 each time. The formula would be

$$a_k = 192 + 250\,(k - 1) \tag{7.1}$$

A pattern seen in four numbers is a guess, not a fact. Here is the proof. "The cube ends in 888" means $n^3 \equiv 888 \pmod{1000}$, where $a \equiv b \pmod{m}$ says that a and b leave the same remainder when divided by m. Since $1000 = 8 \times 125$ and 8 and 125 have no common factor, the condition holds exactly when it holds modulo 8 and modulo 125 separately (this is the Chinese remainder theorem [2]).

- **Modulo 8.** $888 = 111 \times 8$, so we need $n^3 \equiv 0 \pmod 8$. If n is odd, n^3 is odd and not divisible by 8. If n is even, $n = 2m$ and $n^3 = 8m^3$ is divisible by 8. So the condition is: *n is even*.
- **Modulo 125.** $888 = 7 \times 125 + 13$, so we need $n^3 \equiv 13 \pmod{125}$. Trying all 125 residues shows that exactly one works, $n \equiv 67$. (There is also a reason: among the 100 residues not

divisible by 5, cubing is a one-to-one correspondence, because 3 has no common factor with 100 [2]; and a multiple of 5 has a cube divisible by 125, not 13.)

Combining: $n \equiv 67 \pmod{125}$ and n even. The numbers $\equiv 67 \pmod{125}$ are 67,192,317,442, ..., alternately odd and even; the even ones are 192,442,692, ..., which are exactly the numbers $\equiv 192 \pmod{250}$. So the answers are $192{,}192 + 250{,}192 + 500$, ..., and Equation (7.1) is proved. Each query is answered with one multiplication.

7.3.3 How big does it get?

The gotcha is in the size. For the largest k,

$$a_k \le 192 + 250 \times (2 \times 10^{12} - 1) < 5 \times 10^{14} \tag{7.2}$$

far above the largest 32-bit `int`, $2^{31} - 1 \approx 2.1 \times 10^9$, and comfortably below the largest 64-bit `long long`, $2^{63} - 1 \approx 9.2 \times 10^{18}$. Even k itself does not fit in an `int`. **Listing 7.1** does everything in `long long`.

Listing 7.1 `eights.cpp`: the formula of Equation (7.1), computed in 64-bit arithmetic.

```cpp
// EIGHTS (SPOJ): the k-th positive number whose cube ends in 888.
// Those numbers are 192, 442, 692, ... = 192 + 250 (k - 1), k <= 2 * 10^12.
#include <cstdio>

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        long long k;                          // k itself needs 64 bits
        if (std::scanf("%lld", &k) != 1) break;
        std::printf("%lld\n", 192 + 250 * (k - 1));  // k is long long: 64-bit math
    }
    return 0;
}
```

The comment on the `printf` line points at a rule of C++ arithmetic that causes many overflows: the type of an expression is decided by its *operands*, not by the variable that receives it [3]. Because `k` is a `long long`, `250 * (k - 1)` is computed in 64 bits. If `k` were an `int`, the product would be computed in 32 bits and overflow *before* being stored anywhere, even into a `long long` variable. `long long x = a * b;` with two `int` operands is the classic form of this slip; write `(long long)a * b`.

Python's integers have no fixed size, so **Listing 7.2** needs no care at all.

Listing 7.2 `eights.py`: the same formula in Python, whose integers never overflow.

```python
# EIGHTS (SPOJ): the k-th number whose cube ends in 888 is 192 + 250 (k - 1).
# Python integers never overflow, so the formula is all there is.
import sys

data = sys.stdin.buffer.read().split()
t = int(data[0])
print("\n".join(str(192 + 250 * (int(k) - 1)) for k in data[1:1 + t]))
```

On our test file, whose last query is $k = 10^{12}$,

```
4
1
2
3
1000000000000
```

both programs print:

```
192
442
692
249999999999942
```

7.3.4 The same program with `int`

Listing 7.3 is the version a hurried contestant writes.

Listing 7.3 `eights_int.cpp`: the formula with `int`; clean compilation, correct samples, wrong answers for large k.

```
// The same formula with int: compiles cleanly, passes small samples,
// and prints garbage once 250 * (k - 1) no longer fits in 32 bits.
#include <cstdio>

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        int k;                                  // wrong: k can be 2 * 10^12
        if (std::scanf("%d", &k) != 1) break;
        std::printf("%d\n", 192 + 250 * (k - 1));
    }
    return 0;
}
```

```
192
442
692
-1456365626
```

The first three lines are right, and they are exactly the lines a sample in a problem statement tends to contain. The fourth is wrong in two ways at once: `scanf("%d")` cannot store 10^{12} in an `int` (the value it stores is not specified), and the multiplication overflows. g++ compiled the program with `-Wall -Wextra` without a single warning, because nothing in the text is wrong in form; only the values are too large. **Figure 7.1** shows what happens to a 32-bit value that passes its maximum: it jumps to the most negative value and keeps counting from there. For unsigned types that wrap is defined; for `int` it is undefined behavior (Chapter 5), and the negative number is only what usually happens.

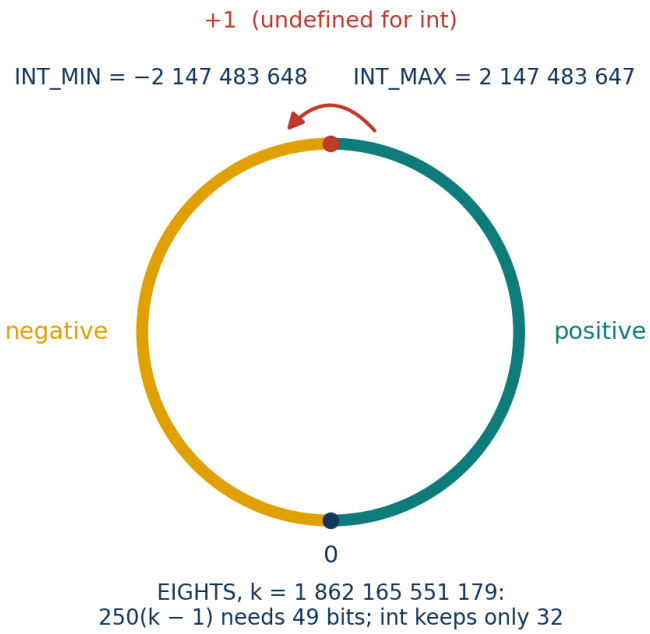


Figure 7.1 A 32-bit `int` as a circle: one step past `INT_MAX` lands on `INT_MIN`; the EIGHTS formula for a large k needs 49 bits.

The symptom, "right on small inputs, wrong and often negative on large ones", is the signature of overflow. When you see a negative number where only positive ones make sense, look for a multiplication of `int`s.

7.3.5 Testing it

The brute force for EIGHTS must not use Equation (7.1); otherwise it would only repeat the program's assumption. **Listing 7.4** walks through 1,2,3, ... and keeps every number whose cube ends in 888, which is the problem statement turned into code.

Listing 7.4 `brute_eights.py`: the brute force for EIGHTS searches upward and needs no formula.

```
# Brute force for EIGHTS: walk through 1, 2, 3, ... and keep every number
# whose cube ends in 888. No formula, no proof needed -- only small k.
import sys

data = sys.stdin.read().split()
t = int(data[0])
wanted = [int(x) for x in data[1:1 + t]]
found, x = [], 0
while len(found) < max(wanted):
    x += 1
    if x ** 3 % 1000 == 888:
        found.append(x)
print("\n".join(str(found[k - 1]) for k in wanted))
```

It is fast for $k \leq 40$, the range of the small generator, and there both `eights` and `eights_int` agree with it on 300 random tests: small tests cannot see the overflow. For large k the second opinion is Equation (7.1) evaluated in Python's exact integers, now justified by the proof and by the small tests together. With the generator's `big` mode, the stress test exposes the `int` version on the first seed:

```
MISMATCH on seed 1
input:
2
1862165551179
1682777914929
expected: 465541387794692 420694478732192
received: 1292646660 -1862878304
```

The lesson is general: **a stress test with small inputs cannot find overflow.** Every stress run needs a second mode that uses the largest values the constraints allow, compared against an independent computation that cannot overflow.

7.4 Beyond Every Built-in Type: FCTRL2

7.4.1 The problem

SPOJ problem FCTRL2, "Small factorials" [4], gives up to 100 values of n with $1 \le n \le 100$ and asks for $n!$ printed in full. The limit of 1 second is generous; the difficulty is only the size. 100! has 158 digits.

7.4.2 How far does `unsigned long long` go?

The largest built-in integer type in standard C++ is `unsigned long long`, 64 bits, with maximum $2^{64} - 1 \approx 1.8 \times 10^{19}$. Since $20! \approx 2.4 \times 10^{18}$ and $21! \approx 5.1 \times 10^{19}$, it holds $n!$ only up to $n = 20$. **Listing 7.5** is the program that forgets this.

Listing 7.5 `fctrl2_ull.cpp`: factorials in the largest built-in type; right up to 20! and silently wrong from 21! on.

```cpp
// FCTRL2 with the biggest built-in type. Right up to 20!, silently wrong
// from 21! on, because unsigned arithmetic wraps around modulo 2^64.
#include <cstdio>

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        int n;
        if (std::scanf("%d", &n) != 1) break;
        unsigned long long f = 1;
        for (int i = 2; i <= n; i++) f *= i;
        std::printf("%llu\n", f);
    }
    return 0;
}
```

With a file that asks for four factorials, of 5, 20, 21 and 30:

```
4
5
20
21
30
```

```
120
2432902008176640000
14197454024290336768
9682165104862298112
```

The first two lines are right. The last two are 21! and 30! reduced modulo 2^{64}: unsigned arithmetic wraps around by definition, so there is no undefined behavior and no warning, only a

wrong number that looks plausible. The stress test of Section 7.4.5 found something even stranger: for $n = 73$ and $n = 98$ the program printed 0. The reason is a small piece of number theory. The number of factors 2 in $n!$ is, by the same layered counting as Legendre's formula for the fives in Chapter 5,

$$v_2(n!) = \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \left\lfloor \frac{n}{8} \right\rfloor + \cdots \tag{7.3}$$

For $n = 66$ that is $33 + 16 + 8 + 4 + 2 + 1 = 64$. So from 66! on, the product is a multiple of 2^{64}, and its remainder modulo 2^{64} is exactly 0. A program that prints 0 for 73! is obviously wrong; one that prints `14197454024290336768` for 21! is not obviously anything, which is why the gotcha survives until the judge finds it.

7.4.3 Digits in an array, as on paper

The fix is to do what a person does with pencil and paper: keep the digits of the number in a list and multiply digit by digit, carrying into the next column. We keep the digits in a `vector<int>` from the *least* significant to the most significant, because then a carry that creates a new leading digit is a `push_back` at the end instead of an insertion at the front.

To multiply the number with digits $d_0, d_1, \ldots, d_{L-1}$ (so the number is $\sum_j d_j\, 10^j$) by a small factor f, walk from d_0 upward with a carry c that starts at 0:

$$x_j = d_j \cdot f + c, \qquad d_j \leftarrow x_j \bmod 10, \qquad c \leftarrow \lfloor x_j / 10 \rfloor \tag{7.4}$$

and when the digits are exhausted, append the remaining carry digit by digit. This is the schoolbook method of Knuth's treatment of multiple-precision arithmetic [5]. With $f \le 100$ and c never exceeding f, every x_j is at most $9 \times 100 + 100 = 1000$, which fits easily in an `int`.

A worked example: multiply 120 (digits 0,2,1, least significant first) by $f = 6$.

- $j = 0$: $x = 0 \times 6 + 0 = 0$, digit 0, carry 0.
- $j = 1$: $x = 2 \times 6 + 0 = 12$, digit 2, carry 1.
- $j = 2$: $x = 1 \times 6 + 1 = 7$, digit 7, carry 0.

The digits are 0,2,7, that is $720 = 6!$. **Listing 7.6** is the program.

Listing 7.6 `fctrl2.cpp`: factorials of up to 158 digits, stored least significant digit first and multiplied as on paper.

```cpp
// FCTRL2 (SPOJ): n! exactly, for n <= 100 (158 digits).
// Digits are kept least significant first in a vector, as on paper.
#include <cstdio>
#include <vector>

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        int n;
        if (std::scanf("%d", &n) != 1) break;
        std::vector<int> digits{1};          // the number 1
        for (int f = 2; f <= n; f++) {
            int carry = 0;
            for (int& d : digits) {          // multiply every digit by f
                int x = d * f + carry;
                d = x % 10;
                carry = x / 10;
            }
            while (carry) { digits.push_back(carry % 10); carry /= 10; }
        }
        for (int i = (int)digits.size() - 1; i >= 0; i--)
            std::putchar('0' + digits[i]);
        std::putchar('\n');
    }
    return 0;
}
```

```
120
2432902008176640000
51090942171709440000
265252859812191058636308480000000
```

The cost: computing $n!$ multiplies a number of at most 158 digits by each factor up to n, so at most 100×158 digit steps per query and 1.6×10^6 for 100 queries. Printing walks the vector backward, most significant digit first.

In Python the problem is one line, because Python's integers are exact at any size and the standard library computes factorials directly; **Listing 7.7** is the whole program.

Listing 7.7 `fctrl2.py`: FCTRL2 in Python, with `math.factorial`.

```python
# FCTRL2 (SPOJ): n! for n <= 100. Python integers are exact at any size.
import math
import sys

data = sys.stdin.read().split()
t = int(data[0])
print("\n".join(str(math.factorial(int(x))) for x in data[1:1 + t]))
```

158 digits are far below Python's 4300-digit conversion limit of Chapter 5, so printing needs no special care here.

7.4.4 The symptom

"Right up to a threshold, then wrong, and sometimes a suspicious 0" is the signature of a value that has outgrown its type. The threshold tells you the type: 12 for factorials in a 32-bit `int` ($13! > 2^{31}$), 20 for 64 bits, 33 for the signed 128-bit extension `__int128`. When a problem's answers can have more than 18 or 19 digits, no built-in type is enough, and you need digits in an array or Python.

7.4.5 Testing it

The brute force computes $n!$ with a plain loop in Python, and the generator draws n from 1 to 100. `fctrl2.cpp` and `fctrl2.py` agreed with it on 300 random tests; `fctrl2_ull.cpp` failed on the first seed, with $n = 73$ and $n = 98$, printing `0 0`.

7.5 Zeros That Must Disappear: ADDREV

7.5.1 The problem

SPOJ problem ADDREV, "Adding Reversed Numbers" [6], plays with numbers written backward. Reversing a number means writing its digits in the opposite order and dropping the zeros that end up in front: the reverse of 1200 is 21, not 0021. The input gives N pairs of numbers, each already *reversed* (so none of them ends in 0, since a reversed number has no zeros to spare at its end). For each pair, the program must reverse both numbers back, add them, and print the reverse of the sum, again without leading zeros.

7.5.2 Where the gotcha lives

The arithmetic is trivial; the zeros are not. Take the pair 4301 and 6, the example of **Figure 7.2**. Reversed, they are 1034 and 6; the sum is 1040; reversed, the sum is "0401", and the correct answer is 401. A program that reverses the string of the sum and prints it prints `0401`, which the judge compares character by character with `401`: *Wrong answer*.

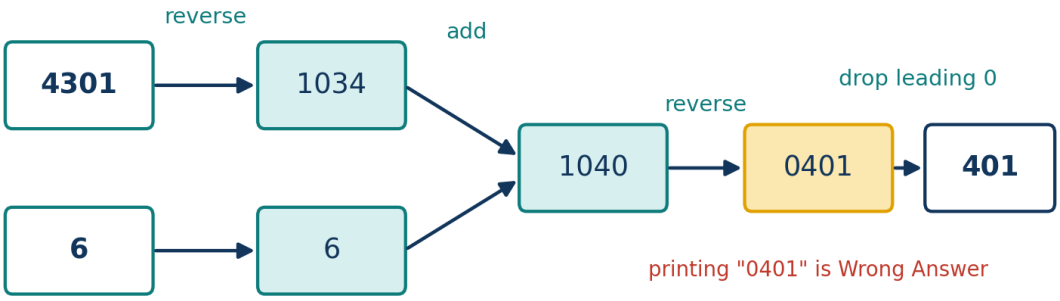


Figure 7.2 ADDREV on the pair 4301 and 6: the sum 1040 reversed is "0401", and the zero in front must be dropped before printing.

There is a second, quieter trap in the input. If we reverse the *strings* 4301 and 6 we get "1034" and "6", which is right. But if a number could end in 0, say "430", its reverse would be "034", and a program that treats it as a string must drop that zero too. The problem promises that inputs do not end in 0, but the sum can: that is exactly the 1040 above. The safe rule is to strip leading zeros after *every* reversal, not only at the end.

7.5.3 The programs

Listing 7.8 works on strings of digits, so it does not care how long the numbers are, and applies one function, `reverseNumber`, everywhere a number is reversed. The addition is the schoolbook addition of Section 7.4, from the last digit to the first, with a carry.

Listing 7.8 `addrev.cpp`: reversals that always drop leading zeros, and schoolbook addition on digit strings.

```cpp
// ADDREV (SPOJ): reverse two numbers, add them, reverse the sum.
// Works on digit strings, so any length is fine; zeros are handled on purpose.
#include <algorithm>
#include <iostream>
#include <string>

// Reverse a digit string and drop the zeros that end up in front.
std::string reverseNumber(std::string s) {
    std::reverse(s.begin(), s.end());
    std::size_t first = s.find_first_not_of('0');
    return first == std::string::npos ? "0" : s.substr(first);
}

// Add two non-negative decimal strings, schoolbook style.
std::string add(const std::string& a, const std::string& b) {
    std::string sum;
    int i = (int)a.size() - 1, j = (int)b.size() - 1, carry = 0;
    while (i >= 0 || j >= 0 || carry) {
        int d = carry;
        if (i >= 0) d += a[i--] - '0';
        if (j >= 0) d += b[j--] - '0';
        sum.push_back(char('0' + d % 10));
        carry = d / 10;
    }
    std::reverse(sum.begin(), sum.end());
    return sum;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n;
    if (!(std::cin >> n)) return 0;
    while (n--) {
        std::string a, b;
        if (!(std::cin >> a >> b)) break;
        std::cout << reverseNumber(add(reverseNumber(a), reverseNumber(b))) << '\n';
    }
    return 0;
}
```

The special case in `reverseNumber` deserves a glance: if the string is all zeros, `find_first_not_of('0')` returns `npos`, and the function returns “0” instead of an empty string. The problem’s inputs are positive, so the case never occurs on the judge, but a function that can return an empty string for a valid number is a gotcha waiting for the next problem.

In Python, **Listing 7.9** lets `int()` do the zero-stripping: converting the reversed string to an integer and back discards leading zeros automatically.

Listing 7.9 `addrev.py`: ADDREV in Python; `int()` of a reversed string drops the leading zeros.

```python
# ADDREV (SPOJ): reverse, add, reverse. int() of a reversed string drops the
# leading zeros that trailing zeros turn into, so no special cases are needed.
import sys


def reverse_number(s):
    return int(s[::-1])


def main():
    data = sys.stdin.read().split()
    n = int(data[0])
    out = []
    for i in range(n):
        a, b = data[1 + 2 * i], data[2 + 2 * i]
        total = reverse_number(a) + reverse_number(b)
        out.append(str(reverse_number(str(total))))
    print("\n".join(out))


if __name__ == "__main__":
    main()
```

Our test file contains the pair of **Figure 7.2**, a pair whose sum does not end in 0, and two pairs whose sums are 1000 and 10:

```
4
4301 6
21 31
999 1
5 5
```

```
401
52
1
1
```

For the pair 999 and 1 the sum is 1000, whose reverse “0001” must be printed as 1; for 5 and 5 the sum is 10, printed as 1.

7.5.4 Testing it

A random generator rarely produces sums that end in several zeros, which are exactly the interesting cases. So `gen_addrev.py` builds 40% of its pairs on purpose: it chooses a round total such as 3000 or 30000 and sets the second number to the total minus the first. The brute force follows the statement literally, one step after another, with lists of characters. Both programs agreed with it on 300 random tests. A version that printed the reversed sum without stripping zeros failed on almost every seed.

The general lesson of ADDREV is about *formats*. Whenever an answer is printed as text, ask what its first and last characters can be: leading zeros, a minus sign on zero (“-0”), trailing spaces, a missing final newline, “1e+06” where “1000000” was expected. Judges compare text, not numbers.

7.6 State That Leaks Between Test Cases: CANDY

7.6.1 The problem

SPOJ problem CANDY, “Candy I” [7], describes packets of candies of different sizes that must be made equal by moving candies from one packet to another, one candy per move. The input

contains several blocks; each starts with the number of packets N ($1 < N \le 10\,000$), followed by the N packet sizes (each below 1000), and the input ends with a line containing -1. For each block, print the smallest number of moves that makes all packets equal, or -1 if it is impossible.

7.6.2 The idea, with a proof

Let the packets hold $c_1, \dots, c_N$ candies, with total $S = \sum c_i$. Moves do not change S, so if all packets end equal, each holds S/N; if N does not divide S, it is impossible and the answer is -1. Otherwise let $A = S/N$. A packet with $c_i > A$ must lose $c_i - A$ candies, and each move takes one candy out of one packet, so at least $\sum_{c_i > A} (c_i - A)$ moves are needed. That many moves also suffice: move candies one by one from any packet above A to any packet below A; since the surpluses and the deficits add up to the same amount, there is always a packet below A while one is above. So

$$\text{moves} = \sum_{i:\, c_i > A} (c_i - A), \qquad A = \frac{S}{N} \tag{7.5}$$

For the block 3, 5, 2, 6: $S = 16$, $A = 4$, the surpluses are $5 - 4 = 1$ and $6 - 4 = 2$, so 3 moves.

7.6.3 The leak

Listing 7.10 is a correct program.

Listing 7.10 `candy.cpp`: the moves of Equation (7.5), with every variable of a block created fresh inside the loop.

```cpp
// CANDY (SPOJ): blocks of packets until a line with -1. For each block,
// the fewest one-candy moves that make all packets equal, or -1.
#include <cstdio>
#include <vector>

int main() {
    int n;
    while (std::scanf("%d", &n) == 1 && n != -1) {
        std::vector<int> packet(n);
        long long total = 0;                 // fresh for every block
        for (int& c : packet) {
            if (std::scanf("%d", &c) != 1) return 0;
            total += c;
        }
        if (total % n != 0) { std::printf("-1\n"); continue; }
        long long target = total / n, moves = 0;
        for (int c : packet)
            if (c > target) moves += c - target;   // every surplus candy moves once
        std::printf("%lld\n", moves);
    }
    return 0;
}
```

Our test file has three blocks: 4, 7, 2 (total 13, not divisible by 3); 3, 5, 2, 6 (the example above); and 1, 9 (total 10, target 5).

```
3
4
7
2
4
3
5
2
6
2
1
9
-1
```

The program prints one line per block:

```
-1
3
4
```

Listing 7.11 differs in one respect: the running total is a global variable, declared once outside the loop, the way many programs declare their arrays.

Listing 7.11 `candy_leak.cpp`: the same program with the total declared outside the loop; every block after the first inherits the candies of the blocks before it.

```
// CANDY with the running total kept outside the loop: block 1 is right,
// every later block inherits the candies of the blocks before it.
#include <cstdio>
#include <vector>

long long total = 0;                          // global: never reset

int main() {
    int n;
    while (std::scanf("%d", &n) == 1 && n != -1) {
        std::vector<int> packet(n);
        for (int& c : packet) {
            if (std::scanf("%d", &c) != 1) return 0;
            total += c;
        }
        if (total % n != 0) { std::printf("-1\n"); continue; }
        long long target = total / n, moves = 0;
        for (int c : packet)
            if (c > target) moves += c - target;
        std::printf("%lld\n", moves);
    }
    return 0;
}
```

On the same file it prints:

```
-1
-1
-1
```

The first answer is right, and only because the first block starts from zero. The second block sees a total of $13 + 16 = 29$, not divisible by 4, and the third $29 + 10 = 39$, not divisible by 2. The stress test found a case where the leak does not even produce -1:

```
MISMATCH on seed 3
input:
6
10
2
5
9
7
9
2
8
4
-1
expected: 7 2
received: 7 0
```

The second block (8 and 4) inherits the 42 candies of the first; the "target" becomes $54/2 = 27$, no packet is above it, and the program confidently prints 0 moves.

The signature of this family is unmistakable once you know it: **the first test case is right and later ones are wrong**, often by amounts that grow. On SPOJ, where most problems pack many test cases into one file, it is among the most common causes of *Wrong answer*. The cures, in order of preference:

1. **Declare per-test variables inside the loop**, as **Listing 7.10** does. They are then created fresh for every test case, and nothing can leak.
2. If a large array must be global (to avoid reallocating it, or because it is too big for the stack), **reset it at the start of each test case**, and reset only the part the test case uses; clearing a 10^6-element array for each of 10^5 small test cases is itself a time-limit problem.
3. Put the whole solution of one test case in a function `solve()` whose locals die when it returns.

The Python version, **Listing 7.12**, follows the first cure and also shows the standard way to read input that ends with a sentinel value: read all tokens at once and walk through them with an index.

Listing 7.12 `candy.py`: CANDY in Python, reading all tokens at once and stopping at the sentinel -1.

```python
# CANDY (SPOJ): read all numbers at once and walk through the blocks.
import sys


def main():
    data = sys.stdin.read().split()
    pos, out = 0, []
    while pos < len(data):
        n = int(data[pos])
        pos += 1
        if n == -1:
            break
        packet = [int(x) for x in data[pos:pos + n]]
        pos += n
        total = sum(packet)                  # fresh for every block
        if total % n:
            out.append("-1")
            continue
        target = total // n
        out.append(str(sum(c - target for c in packet if c > target)))
    print("\n".join(out))


if __name__ == "__main__":
    main()
```

The brute force for CANDY does not use Equation (7.5): it literally moves one candy at a time from the fullest packet to the emptiest until all are equal, and counts the moves. Both correct programs agreed with it on 300 random tests.

7.7 Division, Remainders and Floating Point

7.7.1 Two languages, two answers

What is -7 divided by 2? Mathematics says -3.5; integer division must round it, and C++ and Python round in different directions. **Listing 7.13** and **Listing 7.14** ask the same questions in both languages.

Listing 7.13 `arith_demo.cpp`: integer division, remainder, rounding and floating-point comparison in C++.

```cpp
// Integer division, remainder and floating point: the answers C++ gives.
// Compare with arith_demo.py -- several lines differ.
#include <cmath>
#include <cstdio>

int main() {
    std::printf("-7 / 2   = %d\n", -7 / 2);       // truncates toward zero
    std::printf("-7 %% 2   = %d\n", -7 % 2);      // sign follows the dividend
    std::printf("((-7 %% 5) + 5) %% 5 = %d\n", ((-7 % 5) + 5) % 5);  // safe mod
    double x = 0.1 + 0.2;
    std::printf("0.1 + 0.2 == 0.3 : %s\n", x == 0.3 ? "true" : "false");
    std::printf("0.1 + 0.2 printed with %%.17g : %.17g\n", x);
    bool close = std::fabs(x - 0.3) < 1e-9;          // compare with a tolerance
    std::printf("|x - 0.3| < 1e-9 : %s\n", close ? "true" : "false");
    std::printf("(int)(0.29 * 100) = %d\n", (int)(0.29 * 100));
    std::printf("llround(0.29 * 100) = %lld\n", std::llround(0.29 * 100));
    std::printf("printf %%.2f of 2.675 = %.2f\n", 2.675);
    return 0;
}
```

```
-7 / 2   = -3
-7 % 2   = -1
((-7 % 5) + 5) % 5 = 3
0.1 + 0.2 == 0.3 : false
0.1 + 0.2 printed with %.17g : 0.30000000000000004
|x - 0.3| < 1e-9 : true
(int)(0.29 * 100) = 28
llround(0.29 * 100) = 29
printf %.2f of 2.675 = 2.67
```

Listing 7.14 `arith_demo.py`: the same questions in Python; several answers differ from C++.

```python
# The same questions in Python. Integer division floors, % follows the divisor.
import math

print("-7 // 2  =", -7 // 2)                  # floors toward minus infinity
print("-7 % 2   =", -7 % 2)                   # sign follows the divisor
print("int(-7 / 2) =", int(-7 / 2))           # C++-style truncation, if needed
print("math.fmod(-7, 2) =", math.fmod(-7, 2))
x = 0.1 + 0.2
print("0.1 + 0.2 == 0.3 :", x == 0.3)
print("0.1 + 0.2 printed with repr :", repr(x))
print("math.isclose(x, 0.3) :", math.isclose(x, 0.3))
print("int(0.29 * 100) =", int(0.29 * 100))
print("round(0.29 * 100) =", round(0.29 * 100))
print("round(2.5), round(3.5) =", round(2.5), round(3.5))
print("'%.2f' % 2.675 =", "%.2f" % 2.675)
```

```
-7 // 2  = -4
-7 % 2   = 1
int(-7 / 2) = -3
math.fmod(-7, 2) = -1.0
0.1 + 0.2 == 0.3 : False
0.1 + 0.2 printed with repr : 0.30000000000000004
math.isclose(x, 0.3) : True
int(0.29 * 100) = 28
round(0.29 * 100) = 29
round(2.5), round(3.5) = 2 4
'%.2f' % 2.675 = 2.67
```

Table 7.2 summarizes the differences.

Table 7.2 Integer division, remainder and rounding in C++ and Python.

Expression	C++	Python	Why
−7 divided by 2 (integers)	`-7 / 2` is −3	`-7 // 2` is −4	C++ truncates toward zero, Python floors
remainder of −7 by 2	`-7 % 2` is −1	`-7 % 2` is 1	the remainder follows the quotient
rounding 2.5	`llround(2.5)` is 3	`round(2.5)` is 2	Python rounds halves to even
truncating 0.29 × 100	`(int)` gives 28	`int()` gives 28	0.29 × 100 is slightly below 29

C++ integer division truncates toward zero, and the remainder has the sign of the dividend, so that `(a / b) * b + a % b == a` always holds [3]. Python's `//` rounds down (toward minus infinity), and its `%` has the sign of the divisor, which keeps the same identity with the other rounding [8]. Both are consistent; they are just different. The trap appears in two places.

Remainders of negative numbers. Many problems ask for an answer "modulo m", and a C++ expression such as `(a - b) % m` is negative when $a < b$. The standard repair maps any remainder into $0, 1, \ldots, m-1$:

$$r = ((a \bmod m) + m) \bmod m \tag{7.6}$$

computed with C++'s `%` as the inner and outer mod. For $a = -7$ and $m = 5$: $-7 \,\%\, 5 = -2$, $-2 + 5 = 3$, $3 \,\%\, 5 = 3$, and indeed $-7 = -2 \times 5 + 3$. Python's `%` already gives this result.

Translating between the languages. A formula written for Python, `(x + k) // 2` for a possibly negative `x`, gives a different answer when copied to C++ as `(x + k) / 2`. When negative numbers are possible, check each division.

7.7.2 Floating point

A `double` stores numbers in binary with 53 significant bits, about 16 decimal digits. Most decimal fractions, including 0.1, 0.2 and 0.3, have no exact binary form, just as 1/3 has no exact decimal form. So `0.1 + 0.2` is the double nearest to 0.30000000000000004, and it is not equal to the double nearest to 0.3: both programs print `false` [9]. The consequences for contest code are three rules.

1. **Never compare computed doubles with `==`.** Compare with a tolerance ε, as the demo does:

$$|x - y| < \varepsilon \tag{7.7}$$

with ε chosen from the problem: if answers are checked to 10^{-6}, compute with $\varepsilon = 10^{-9}$. Python offers `math.isclose`, which uses a relative tolerance.

2. **Do not truncate a double that should be an integer.** 0.29×100 is 28.999999999999996, and `(int)` or `int()` cuts it to 28. Round to the nearest integer instead, with `llround` or `round`. Better still, avoid the double altogether: prices in rupiah, or amounts in cents, belong in integers.
3. **Know how your output is rounded.** `printf("%.2f", 2.675)` prints 2.67, not 2.68, because the stored double is slightly below 2.675. When the judge compares with a tolerance, this does not matter; when it compares text exactly, compute in integers or add the rounding yourself.

Chapter 5's warning about `-ffast-math` belongs here too: with that flag, even the tolerance test can be optimized in surprising ways.

7.8 Off-by-One and the Shape of the Input

7.8.1 Counting posts

Mas Dedi's fence has a precise mathematical form. The number of integers in the closed range from L to R, both included, is

$$|\{L, L+1, \ldots, R\}| = R - L + 1 \tag{7.8}$$

not $R - L$: ten meters of fence from post 0 to post 10 need $10 - 0 + 1 = 11$ posts. Almost every off-by-one error is a confusion between *closed* ranges $[L, R]$, which include both ends, and *half-open* ranges $[L, R)$, which include L and exclude R, with $R - L$ elements. C++ and Python both prefer half-open ranges: `for (int i = 0; i < n; i++)`, `range(0, n)`, `v.begin()` to `v.end()`, `s[a:b]`. Most problem statements use closed ranges: "print all primes p with $m \le p \le n$". The translation is where the slip happens.

Three habits prevent most of these errors.

- **Write the range in words before writing the loop**: "from m to n, both included". Then `x <= n`, not `x < n`. The PRIME1 program of Chapter 6 does exactly this.
- **Test the ends.** Every range has a first and a last element; the smallest input (1 element, $L = R$) and the largest are where off-by-one errors show.
- **Keep one convention inside your program.** If you use half-open ranges everywhere, as the Python merge sort of Chapter 8 does with its halves, the lengths are always `hi - lo` and the middle splits cleanly.

7.8.2 The input is not always what it seems

Several gotchas live in reading the input.

- **The number of test cases.** Some problems give t first, some end with a sentinel (-1 in CANDY, 42 in SPOJ's classic first problem TEST), some simply end at the end of the file. Reading the wrong way either misses tests or reads garbage. In C++, `while (cin >> x)` and `while (scanf("%d", &x) == 1)` stop cleanly at the end of the file; in Python, reading all tokens at once, as every Python program in this chapter does, makes all three shapes easy.
- **Blank lines and spacing.** INVCNT in Chapter 8 separates test cases with empty lines. Token-based reading (`cin >>`, `scanf`, `split()`) skips them automatically; line-based reading (`getline`, `input()`) does not. Prefer tokens unless the problem is about lines.
- **Line endings.** A file created on Windows ends its lines with the two characters `\r\n`. A program that reads lines with `getline` on Linux keeps the `\r` at the end of each line, and a comparison such as `line == "YES"` fails. Judges generate their files on Linux, but your own test files, typed on Windows, may not be; token-based reading ignores the problem.
- **The last line.** Some files end without a final newline. `getline` still returns the last line; code that reads character by character until `'\n'` may loop forever or lose it.

7.8.3 The shape of the output

The judge compares your output with the expected output, usually token by token, sometimes character by character. The output gotchas are the mirror image of the input ones: a missing blank line between test cases (PRIME1 requires one), "Case 1:" where the statement says "Case #1:", `YES` where it says `Yes`, `-0.00` where it expects `0.00`. The statement's output format is a specification, not a suggestion; read it twice.

7.8.4 A comparator that says "less or equal"

One more slip belongs with the off-by-one family, because it is the same confusion between $<$ and $\leq$ in a new place. `std::sort` accepts a comparison function, and a contestant who wants to sort pairs by their first element writes

```
sort(v.begin(), v.end(), [](const P& a, const P& b) { return a.first <= b.first; });
```

The C++ standard requires the comparison to behave like "strictly less than": for equal elements it must answer *false* in both directions [3]. A comparator with `<=` answers *true* for `cmp(x, x)`, which breaks that rule, and the program's behavior is undefined. In practice, with many equal keys, the sort can read outside the array and crash, or leave the elements in a wrong order. The fix is one character, `<` instead of `<=`; if equal first elements must be ordered by something else, compare the second element when the first ones are equal. Python's `sorted` takes a `key` function instead of a comparison and has no such trap, which is one more reason to prefer `key=` when a Python program sorts records.

The slip is sly because it rarely shows on small inputs: with few equal keys, sorting implementations often happen to give the right result. It appears on large inputs with many duplicates, exactly the kind of input the judge has and a sample rarely has. A stress test whose generator deliberately creates many equal keys, for example values drawn only from 1 to 3, finds it far sooner than a generator with widely spread values.

7.9 Testing It

Every problem of this chapter was stress-tested in the same way: a generator, a brute force that follows the statement without cleverness, and the harness `stress.py` of Chapter 8, 300 random tests per run. **Table 7.3** records the results for the correct programs and for the deliberately broken ones.

Table 7.3 Stress-test results for the programs of this chapter (300 random tests per run).

Program	Generator	Result
`eights.cpp`, `eights.py`	small k and large k	OK: 300 of 300, both modes
`eights_int.cpp`	small k	OK: 300 of 300 (overflow invisible)
`eights_int.cpp`	large k	mismatch on seed 1
`fctrl2.cpp`, `fctrl2.py`	$1 \le n \le 100$	OK: 300 of 300
`fctrl2_ull.cpp`	$1 \le n \le 100$	mismatch on seed 1 (prints 0 for 73!)
`addrev.cpp`, `addrev.py`	pairs with round sums	OK: 300 of 300
`candy.cpp`, `candy.py`	1 to 4 blocks	OK: 300 of 300
`candy_leak.cpp`	1 to 4 blocks	mismatch on seed 3

Two rows deserve emphasis. The `eights_int` row with small k passes completely: a stress test finds only what its generator can produce. And every broken program was caught within the first three seeds, which is typical: gotchas are not rare events, they are systematic, and a few hundred random tests are usually enough to expose them.

7.10 Using It for Real

The seven families of **Table 7.1** make a checklist that takes two minutes before any submission.

1. **Sizes.** What is the largest value any variable, sum or product can reach? Is every one of them in a type that holds it, and is every multiplication done in that type?
2. **Beyond types.** Can an answer have more than 18 digits? Then digits in an array, or Python.
3. **Formats.** What are the first and last characters of every printed value? Leading zeros, signs, precision, separators, final newline.
4. **State.** Is every per-test variable created inside the test loop, or reset at its start?
5. **Division.** Can anything negative be divided or reduced modulo m?
6. **Floating point.** Is any `==` applied to doubles? Is any double truncated to an integer?
7. **Ranges and input.** Closed or half-open? What are the smallest and largest inputs, and what does the program do at each end? How does the input end?

The same list serves anyone who turns rules into a program. The grocery that computes a 2.5% discount on a price in rupiah meets families 5 and 6 at once: $2.5\% \times 19\,900 = 497.5$, and whether the customer pays 19,402 or 19,403 depends on a rounding rule that the program must state rather than inherit from `round`. A payroll sheet that adds a year of salaries for a hundred

employees meets family 1 in any 32-bit column. And a cooperative's monthly report that starts each month from last month's totals by accident meets family 4. None of these is an algorithm problem; all of them are real.

One more habit makes the list your own. Keep a short notebook, on paper or in a text file next to your solutions, and write one line every time a submission fails for a reason from this chapter: the problem, the family, the line that was wrong. After a few months the notebook shows which families catch *you*, and those deserve the first look in every checklist. Experienced contestants are not people who never make these slips; they are people who know which ones they make.

7.11 Common Mistakes, and How to Avoid Them

This chapter is itself a catalogue of mistakes, so the list here is short: the meta-mistakes that let the others through.

- **Trusting the samples.** Samples are chosen to explain the problem, not to test your program; they are small and friendly. EIGHTS' samples are the kind of input `eights_int.cpp` passes.
- **Stress-testing only small inputs.** Overflow and performance problems appear only at the top of the constraints. Always add a "big" mode to the generator.
- **Letting the brute force share the fast program's assumption.** A brute force for EIGHTS that uses Equation (7.1) checks nothing. Make the brute force follow the statement.
- **Fixing a symptom instead of the family.** After finding one overflow, check every other multiplication in the program; after finding one leak, check every global.
- **Ignoring the verdict's pattern.** "Wrong answer on test 1" and "wrong answer on test 17" are different clues: the first usually means a format or reading problem, the second a size or edge problem.

7.12 Chapter Summary

- A correct idea can still give *Wrong answer* through seven families of slips (**Table 7.1**), each with a recognizable symptom.
- EIGHTS: the numbers whose cube ends in 888 are $192 + 250(k-1)$ (Equation (7.1)), proved by splitting 1000 into 8×125; the answer reaches 5×10^{14} (Equation (7.2)), so every step must be 64-bit, and the type of a product is the type of its operands.
- FCTRL2: `unsigned long long` holds factorials only to 20! and prints 0 from 66! on (Equation (7.3)); digits in an array, multiplied with a carry (Equation (7.4)), handle 158 digits, and Python needs no care.
- ADDREV: every reversal must drop leading zeros; the output "0401" instead of "401" is a *Wrong answer*.
- CANDY: the moves are the total surplus over the average (Equation (7.5)); a total declared outside the test loop leaks into later blocks, the classic "first case right, later wrong" symptom.
- C++ division truncates and Python floors; remainders of negative numbers need Equation (7.6); doubles are compared with a tolerance (Equation (7.7)) and rounded, not truncated.
- Ranges from L to R inclusive have $R - L + 1$ elements (Equation (7.8)); input and output formats are specifications.

7.13 Exercises

1. Without running anything, find the 5th and the 1000th number whose cube ends in 888, and check that a_{1000}^3 ends in 888 by computing only its last three digits.
2. Which numbers have a cube ending in 111? Use the method of Section 7.3.2 and give the formula for the k-th one.
3. The expression `long long area = w * h;` with `int w = 100000, h = 100000;` prints a wrong area. Give two different one-line fixes and explain why `long long area = (long long)(w * h);` is not one of them.
4. Change `fctrl2.cpp` to store *four* decimal digits per `int` (base 10,000) instead of one. What is the largest intermediate value in Equation (7.4) now, and how must the printing change?
5. A problem asks for the sum of n numbers modulo $10^9 + 7$, where the numbers can be negative (down to -10^9). Write the accumulation loop in C++ so that the printed result is always in $0, \ldots, 10^9 + 6$.
6. A program reads a list of prices such as `19.90` as doubles, multiplies each by 100 and truncates to get cents. For which of the prices 0.29, 0.57, 1.15, 19.90 does it give a wrong number of cents? Suggest a way to read the prices that cannot go wrong.

7.14 Solutions to the Exercises

1. By Equation (7.1), $a_5 = 192 + 250 \times 4 = 1192$ and $a_{1000} = 192 + 250 \times 999 = 249\,942$. Only the last three digits matter for the last three digits of the cube: $249\,942 \equiv 942 \pmod{1000}$, and $942^2 = 887\,364 \equiv 364$, then $364 \times 942 = 342\,888 \equiv 888$. So a_{1000}^3 ends in 888, as it must.

2. Modulo 8: $111 \equiv 7 \pmod 8$, and cubing permutes the odd residues modulo 8 ($1^3 = 1$, $3^3 = 27 \equiv 3$, $5^3 = 125 \equiv 5$, $7^3 = 343 \equiv 7$), so $n \equiv 7 \pmod 8$. Modulo 125: 111 is not divisible by 5, so by the argument of Section 7.3.2 there is exactly one solution of $n^3 \equiv 111 \pmod{125}$; trying the residues gives $n \equiv 96$, and indeed $96^3 = 884\,736 = 7077 \times 125 + 111$. Combining $n \equiv 7 \pmod 8$ and $n \equiv 96 \pmod{125}$ gives $n \equiv 471 \pmod{1000}$: $471^3 = 104\,487\,111$. So the numbers are 471,1471,2471, ..., and the k-th is $471 + 1000(k-1)$. Unlike 888, the period is the full 1000, because modulo 8 there is only one solution instead of four.

3. `w * h` is computed in `int` because both operands are `int`, overflows, and only then is converted. Fixes: `long long area = (long long)w * h;` (one operand converted before the multiplication makes the whole product 64-bit), or declare `w` and `h` as `long long`. The cast `(long long)(w * h)` converts the result *after* the overflowing multiplication, too late.

4. With base $B = 10\,000$, Equation (7.4) becomes $x_j = d_j f + c$, $d_j = x_j \bmod B$, $c = \lfloor x_j / B \rfloor$. With $d_j \le 9999$, $f \le 100$ and $c \le 100$, the largest x_j is $9999 \times 100 + 100 = 1\,000\,000$, which fits in an `int`. Printing: the most significant "digit" is printed normally, every other one with exactly four digits including leading zeros, `printf("%04d", d)`; forgetting the zero padding is a new ADDREV-style gotcha. The loop does a quarter of the steps.

5. Apply Equation (7.6) after every addition:

```
const long long MOD = 1000000007;
long long sum = 0;
for (long long x : values)
    sum = ((sum + x) % MOD + MOD) % MOD;   // the safe remainder, at every step
```

`sum + x` is at least -10^9 and at most about 2×10^9, which fits in `long long`; the inner `%` can be negative, and adding `MOD` before the outer `%` makes it non-negative.

6. All four. 0.29×100 gives 28.999999999999996 and truncates to 28; $0.57 \times 100 = 56.99999999999999$ gives 56; $1.15 \times 100 = 114.99999999999999$ gives 114; and $19.90 \times 100 = 1989.9999999999998$ gives 1989. Each is one cent short. Reading safely: round instead of truncating (`llround(x * 100)`), or better, never make a double at all: read the price as text, split at the decimal point, and compute `rupiah * 100 + cents` in integers.

References

[1] Sphere Online Judge, "EIGHTS - Triple Fat Ladies." [Online]. Available: https://www.spoj.com/problems/EIGHTS/ (accessed Sep. 2026).

[2] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, 6th ed., revised by D. R. Heath-Brown and J. H. Silverman. Oxford, UK: Oxford University Press, 2008.

[3] cppreference.com, "Arithmetic operators." [Online]. Available: https://en.cppreference.com/w/cpp/language/operator_arithmetic (accessed Sep. 2026).

[4] Sphere Online Judge, "FCTRL2 - Small factorials." [Online]. Available: https://www.spoj.com/problems/FCTRL2/ (accessed Sep. 2026).

[5] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[6] Sphere Online Judge, "ADDREV - Adding Reversed Numbers." [Online]. Available: https://www.spoj.com/problems/ADDREV/ (accessed Sep. 2026).

[7] Sphere Online Judge, "CANDY - Candy I." [Online]. Available: https://www.spoj.com/problems/CANDY/ (accessed Sep. 2026).

[8] Python Software Foundation, "Built-in Types: Numeric Types," *The Python Standard Library*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

[9] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, Mar. 1991.

Chapter 8

Debugging TLE and WA: Stress Testing and Brute-Force Checkers

A judge tells you almost nothing. It does not show the test that failed, the answer it expected, or the line that was slow; it says *Wrong answer* or *Time limit exceeded*, and waits. This chapter is about finding the bug anyway, on your own machine, before or after the verdict. By the end of it you will be able to write the three small programs of a stress test, a generator, a brute force and a harness, and use them to find a wrong answer in minutes; shrink a failing input until the bug is obvious; build worst-case inputs and measure how a program's time grows, to predict a time limit before the judge does; use assertions, the debug mode of the C++ library and the address sanitizer to stop a program at the line where it goes wrong; and apply all of this to two real SPOJ problems, INVCNT and PALIN, each with a bug that we plant, find and fix in front of you.

8.1 Counting the Money Twice

Bu Endang is the treasurer of an *arisan*, a rotating savings group of thirty women in a village near Sleman. Every month each member pays in, one member takes the whole pot home, and Bu Endang must make sure that the pot is exactly right. She counts it twice, in two different ways. First she adds up the payments in her notebook with a calculator, member by member. Then she sorts the banknotes on the table by denomination, counts each pile, and multiplies: so many hundred-thousands, so many fifty-thousands, and so on. If the two totals agree, she is confident; two different methods rarely make the same mistake. If they disagree, she does not recount everything. She splits the notebook in two halves and the notes in the same two groups of members, finds the half where the totals differ, and splits again, until she is looking at one member's payment and one small pile of notes.

Bu Endang has invented, without the name, the method of this chapter. The notebook total is your fast program, the one you want to submit. The pile count is a *brute force*, a second method so simple that it is almost certainly right, even if it is slow. Comparing the two on many inputs is a *stress test*. Splitting the disagreement in halves until it is small enough to see is *shrinking*. And the reason it works is the same for her and for you: an error in one method is unlikely to be repeated, by coincidence, in a completely different method.

Programmers need the method more than treasurers do, because the judge gives even less information than a wrong pot. It never shows the failing test. A contestant who gets *Wrong answer* and simply rereads the code is like a treasurer who recounts the notebook with the same calculator: she will make the same mistake again. A second method finds it.

8.2 The Idea in Plain Words

> **DEFINITION.** A *candidate* is the program you want to submit. A *brute force* (or reference solution) is a program for the same problem that is written to be obviously correct rather than fast, for example by trying every possibility. A *generator* is a program that writes a random valid input, determined by a number called the *seed*, so that the same seed always gives the

> same input. A *stress test* runs the generator for many seeds and compares the candidate's output with the brute force's on each input, stopping at the first difference. A *minimal failing input* is a failing input that cannot be made smaller and still fail.

Why should small random inputs find bugs that the judge's large tests find? Because most bugs are not about size. An off-by-one, a missing carry, a wrong comparison or a forgotten special case fails on *some* small input as well, typically one with two or three elements or two or three digits, and a few hundred random small inputs contain most of the small shapes. This observation has a name in software engineering, the *small scope hypothesis*: most bugs can be demonstrated on small instances [1]. Experienced contestants rely on it daily, and guides to competitive programming recommend exactly this routine of a slow reference solution and random tests [2]. The exceptions are exactly the bugs of size: overflow, which needs large values (Chapter 7), and slowness, which needs large inputs. For those, the stress test needs a second kind of input, the largest and the worst the constraints allow, and a way to measure time. This chapter builds both.

Table 8.1 summarizes what each verdict suggests and which tool this chapter uses for it.

Table 8.1 What each verdict usually means, and the first tool to reach for.

Verdict	Most likely causes	First tool
Wrong answer, small tests pass	a logic slip on some small shape	random stress test (Sections 8.3 to 8.5)
Wrong answer, only large tests	overflow, a size-dependent branch	largest inputs against an independent formula (Section 8.5.3)
Time limit exceeded	wrong complexity, or a hidden quadratic step	timing on growing and worst-case inputs (Sections 8.6, 8.7.4)
Runtime error	index out of range, stack overflow, division by zero	debug mode, sanitizers, assertions (Section 8.8)

8.3 The Harness

The three programs of a stress test are small, and the harness is the smallest. **Listing 8.1** is the one used throughout this book. It takes three commands, runs the generator with seeds 1, 2, 3, …, feeds each input to the brute force and to the candidate, and compares their outputs token by token.

Listing 8.1 `stress.py`: runs a generator, a brute force and a candidate on many seeds and stops at the first disagreement.

```python
# Stress tester: run a generator, a brute force and a candidate on many
# random inputs and stop at the first disagreement.
# Usage: python3 stress.py GEN BRUTE CAND [iterations] [first_seed]
#   GEN, BRUTE, CAND are commands in quotes, e.g. "python3 gen.py" "./fast".
#   The generator receives the seed as its last argument.
import shlex
import subprocess
import sys


def run(cmd, text_in=""):
    try:
        r = subprocess.run(shlex.split(cmd), input=text_in, capture_output=True,
                           text=True, timeout=5)
    except subprocess.TimeoutExpired:
        return "TIMEOUT (more than 5 s)"
    if r.returncode != 0:
        raise RuntimeError("%s exited with %d\n%s" % (cmd, r.returncode, r.stderr))
    return r.stdout


def main():
    gen, brute, cand = sys.argv[1:4]
    iterations = int(sys.argv[4]) if len(sys.argv) > 4 else 300
    first = int(sys.argv[5]) if len(sys.argv) > 5 else 1
    for seed in range(first, first + iterations):
        test = run("%s %d" % (gen, seed))
        want = run(brute, test).split()
        got = run(cand, test).split()
        if want != got:
            print("MISMATCH on seed %d" % seed)
            print("input:\n" + test.rstrip())
            print("expected: " + " ".join(want)[:200])
            print("received: " + " ".join(got)[:200])
            with open("failing_input.txt", "w") as f:
                f.write(test)
            sys.exit(1)
    print("OK: %d random tests agree" % iterations)


if __name__ == "__main__":
    main()
```

A few decisions in it are deliberate.

- **Commands, not languages.** The three arguments are shell-style commands, split by `shlex.split`, so any mixture works: a Python generator, a C++ brute force and a Python candidate, or the other way round.
- **Seeds, not randomness.** The generator receives the seed as its last argument and must produce the same input for the same seed. A failing case can then be reproduced by its seed alone, which is also how a failure is shared with a friend.
- **Token comparison.** `split()` ignores differences in spaces and line breaks, as most judges do. For a problem whose judge compares exact text, compare `stdout` directly instead.
- **A timeout.** A candidate that loops forever would hang the harness; after 5 seconds the run is recorded as `TIMEOUT` and counted as a disagreement. Chapter 5 used this to catch an endless loop.
- **The failing input is saved** in `failing_input.txt`, ready to be fed to a debugger or to the candidate with extra printing.

Run it from a terminal in the chapter's `tests` folder, in VS Code or anywhere else: `python3 stress.py "python3 gen.py" build/brute build/fast 300` on Linux and macOS, `py stress.py "py gen.py" build\brute.exe build\fast.exe 300` on Windows.

A good generator follows four rules. It writes only *valid* inputs, exactly as the statement describes them, or the comparison is meaningless. It keeps them *small*, so that the brute force is fast and a failure is readable. It lets the size be chosen, so that a failure can be shrunk. And it aims at the *dangerous shapes*: repeated values, values at the ends of the ranges, sorted and reversed orders, digits 9 and 0. **Listing 8.2** is the generator for the first problem of the chapter.

Listing 8.2 `gen_invcnt.py`: random INVCNT inputs with distinct values and the blank lines of the real input format; the largest array size can be chosen.

```
# Random INVCNT input: distinct values, tiny arrays, blank lines as on SPOJ.
# Usage: python3 gen_invcnt.py [max_n] seed   (max_n defaults to 6)
import random
import sys

rng = random.Random(int(sys.argv[-1]))
max_n = int(sys.argv[1]) if len(sys.argv) > 2 else 6
t = rng.randint(1, 3)
print(t)
for _ in range(t):
    n = rng.randint(1, max_n)
    print()
    print(n)
    print("\n".join(map(str, rng.sample(range(1, 20), n))))
```

It writes the empty line before each test case that INVCNT's input format contains (Section 8.4.1), so that the candidate's reading is tested too, and it takes an optional first argument `max_n`, which Section 8.5.2 uses to shrink a failure.

8.3.1 Checking the generator itself

A generator is a program too, and it can be wrong. A generator that occasionally writes an invalid input, two equal values in INVCNT or a number with a leading zero in PALIN, produces "failures" that are not bugs of the candidate at all, and a contestant can lose an hour debugging a correct program. Two cheap habits prevent this.

First, **read a few generated inputs** before the first stress run: `python3 gen_invcnt.py 1`, `python3 gen_invcnt.py 2`, and so on. Five seeds, ten seconds, and most mistakes in a generator are visible to the eye: a missing blank line, a count n that does not match the number of values, a value outside the allowed range.

Second, for anything less obvious, write a **validator**: a short program that reads an input and checks every constraint of the statement, printing nothing when the input is valid and a message when it is not. For INVCNT it checks that the number of test cases matches t, that each array has exactly n values, and that the values are distinct positive integers up to 10^7. Running the validator on a few hundred generated inputs, in the same loop as the stress test, costs a few seconds. Problem setters validate their own test files in exactly this way; a contestant who does the same never chases a bug that the input invented.

A validator has a second use. When a stress test fails, the failing input has already been checked, so the difference must come from one of the two programs; and when the brute force and the candidate disagree on an input that the validator rejects, the fault is in the generator. Either way, the question "whose fault is it?" has a quick answer.

8.4 INVCNT: Counting Inversions

8.4.1 The problem

SPOJ problem INVCNT, "Inversion Count" [3], gives an array A of n distinct positive integers, with $n \le 200\,000$ and values up to 10^7, and asks how many *inversions* it has: pairs of positions $i < j$ with $A[i] > A[j]$, that is, pairs that are in the wrong order. The input starts with the number of test cases t; each test case is preceded by an empty line and gives n followed by the n values, one per line. The time limit is about 3.6 seconds.

For the array 7,2,9,4 the inversions are (7,2), (7,4) and (9,4), so the answer is 3. A sorted array has no inversions; an array in reverse order has an inversion for every pair, the most possible:

$$I_{\max}(n) = \binom{n}{2} = \frac{n(n-1)}{2} \tag{8.1}$$

For $n = 200\,000$ that is 19 999 900 000, almost ten times the largest `int`. The counter must be 64-bit.

8.4.2 The brute force

The brute force checks every pair, exactly as the definition says. **Listing 8.3** is the program.

Listing 8.3 `invcnt_brute.cpp`: the brute force looks at every pair; it is obviously right and takes quadratic time.

```
// INVCNT brute force: look at every pair (i, j) with i < j. O(n^2) time,
// obviously correct -- the reference the fast program is checked against.
#include <cstdio>
#include <vector>

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        int n;
        if (std::scanf("%d", &n) != 1) break;
        std::vector<long long> a(n);
        for (auto& x : a)
            if (std::scanf("%lld", &x) != 1) return 0;
        long long inversions = 0;
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j < n; j++)
                if (a[i] > a[j]) inversions++;
        std::printf("%lld\n", inversions);
    }
    return 0;
}
```

It makes $\binom{n}{2}$ comparisons, about 2×10^{10} for the largest array, which is far too slow for the judge but instant for the arrays of at most 6 elements that the generator writes. It is a good brute force because there is nothing to get wrong: two loops and a comparison, straight from the definition.

8.4.3 The fast idea: count while merging

Merge sort sorts an array by sorting its two halves recursively and then *merging* them: repeatedly compare the smallest remaining element of each half and move the smaller one to the output [4]. The inversions can be counted during the merge at no extra cost.

Every inversion (i, j) is of one of three kinds: both positions in the left half, both in the right half, or i in the left and j in the right. The first two kinds are counted by the recursive calls. Sorting a half does not change which elements are in it, so it does not change the number of *cross* inversions, which we count while merging. Suppose the left half is sorted as $L[\text{lo}..\text{mid}]$, the right half as $R[\text{mid}+1..\text{hi}]$, and the merge is about to take $R[j]$ because it is smaller than $L[i]$, the smallest element still waiting in the left half. Then every element still waiting in the left half, $L[i], L[i+1], \ldots, L[\text{mid}]$, is larger than $R[j]$ (they are at least $L[i]$), and every left element already taken is smaller than $R[j]$ (it was taken because it was at most some right element no larger than $R[j]$). So $R[j]$ forms a cross inversion with exactly the waiting left elements, and the count increases by

$$\text{mid} - i + 1 \tag{8.2}$$

the number of elements in $L[i..\text{mid}]$, closed at both ends. Each right element is taken exactly once, so every cross inversion is counted exactly once.

The time satisfies the merge-sort recurrence: two halves plus a linear merge,

$$T(n) = 2\,T(n/2) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n \log n) \tag{8.3}$$

about $200\,000 \times 18 \approx 3.6 \times 10^6$ steps for the largest array. **Listing 8.4** is the program.

Listing 8.4 `invcnt.cpp`: inversion counting by merge sort, with a 64-bit counter.

```cpp
// INVCNT (SPOJ): count pairs i < j with a[i] > a[j], n <= 200,000.
// Merge sort: while merging two sorted halves, an element taken from the
// right half jumps over every element still waiting in the left half.
#include <cstdio>
#include <vector>

long long sortCount(std::vector<int>& a, std::vector<int>& tmp, int lo, int hi) {
    if (hi - lo < 1) return 0;                    // one element: no pairs
    int mid = lo + (hi - lo) / 2;                 // halves [lo, mid], [mid+1, hi]
    long long count = sortCount(a, tmp, lo, mid) + sortCount(a, tmp, mid + 1, hi);
    int i = lo, j = mid + 1, k = lo;
    while (i <= mid && j <= hi) {
        if (a[i] <= a[j]) {
            tmp[k++] = a[i++];
        } else {
            count += mid - i + 1;                 // a[i..mid] are all > a[j]
            tmp[k++] = a[j++];
        }
    }
    while (i <= mid) tmp[k++] = a[i++];
    while (j <= hi) tmp[k++] = a[j++];
    for (k = lo; k <= hi; k++) a[k] = tmp[k];
    return count;
}

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        int n;
        if (std::scanf("%d", &n) != 1) break;
        std::vector<int> a(n), tmp(n);
        for (int& x : a)
            if (std::scanf("%d", &x) != 1) return 0;
        std::printf("%lld\n", n > 0 ? sortCount(a, tmp, 0, n - 1) : 0LL);
    }
    return 0;
}
```

The recursion depth is only $\log_2 n \approx 18$, so the stack of Chapter 6 is no concern. **Listing 8.5** does the same bottom-up in Python: it merges runs of width 1, 2, 4, ... until one run covers the array, with half-open ranges `[lo, mid)` and `[mid, hi)`, so the count added is `mid - i`, the Python form of Equation (8.2).

Listing 8.5 `invcnt.py`: bottom-up merge sort that counts inversions, without recursion.

```python
# INVCNT (SPOJ) in Python: a bottom-up merge sort that counts inversions.
# No recursion, so no recursion limit; about 18 passes for n = 200,000.
import sys


def count_inversions(a):
    n, width, count = len(a), 1, 0
    src, dst = a[:], [0] * len(a)
    while width < n:
        for lo in range(0, n, 2 * width):
            mid, hi = min(lo + width, n), min(lo + 2 * width, n)
            i, j, k = lo, mid, lo
            while i < mid and j < hi:
                if src[i] <= src[j]:
                    dst[k] = src[i]
                    i += 1
                else:
                    count += mid - i          # src[i:mid] all jump over src[j]
                    dst[k] = src[j]
                    j += 1
                k += 1
            dst[k:hi] = src[i:mid] if i < mid else src[j:hi]
        src, dst = dst, src
        width *= 2
    return count


def main():
    data = sys.stdin.buffer.read().split()
    t, pos, out = int(data[0]), 1, []
    for _ in range(t):
        n = int(data[pos])
        a = list(map(int, data[pos + 1:pos + 1 + n]))
        pos += 1 + n
        out.append(str(count_inversions(a)))
    print("\n".join(out))


if __name__ == "__main__":
    main()
```

Our sample has two test cases, the array of Section 8.4.1 and an array of six values in reverse order, which must give $\binom{6}{2} = 15$:

```
2

4
7
2
9
4

6
60
50
40
30
20
10
```

```
3
15
```

The brute force and the Python program print the same two numbers.

8.5 Hunting a Wrong Answer

8.5.1 A program with two slips

Programs that are written in a hurry rarely look like **Listing 8.4** the first time. The file `invcnt_bug.cpp` in the chapter's folder is the same program with two slips that we have seen many contestants make: the counter and the function return an `int`, and the count added at each step is `mid - i` instead of `mid - i + 1`, the half-open form of Equation (8.2) used by mistake on closed ranges. The differences from **Listing 8.4**, as `diff` shows them:

```
<     count += mid - i + 1;       // a[i..mid] are all > a[j]
---
>     count += mid - i;           // slip: forgot the + 1

<     long long sortCount(...)    // and: long long count = ...
---
>     int sortCount(...)          // and: int count = ...
```

On the sample it prints `0` and `8` instead of 3 and 15. On a real judge you would see only *Wrong answer*.

8.5.2 Finding it and shrinking it

The stress test finds the difference on the very first seed:

```
python3 stress.py "python3 gen_invcnt.py" build/invcnt_brute build/invcnt_bug 300
```

```
MISMATCH on seed 1
input:
1

5
3
9
4
16
13
expected: 2
received: 0
```

The array 3,9,4,16,13 is already small, but we can do better, and smaller inputs make the cause easier to see. The generator's first argument limits the size; with arrays of at most 2 elements,

```
python3 stress.py "python3 gen_invcnt.py 2" build/invcnt_brute build/invcnt_bug 300
```

```
MISMATCH on seed 6
input:
3

1
16

2
2
1

1
19
expected: 0 1 0
received: 0 0 0
```

the second test case, the array 2,1, has one inversion, and the program reports none. Now trace it by hand: the halves are [2] and [1], so lo = mid = 0 and hi = 1; the merge compares 2 with 1, takes

1 from the right half, and adds `mid - i` $= 0 - 0 = 0$. The waiting left part $L[0..0]$ has one element, not zero: the formula is off by one. With the $+1$ restored, the stress test runs clean:

```
OK: 300 random tests agree
```

This is the whole method in miniature: a random failure, shrunk to the smallest shape, traced by hand. Shrinking by limiting the size is the simplest strategy; another is to delete elements from the failing input one at a time, keeping each deletion that still fails, until no single deletion keeps the failure. Some testing tools do this automatically; by hand, it rarely takes more than a few minutes.

8.5.3 The bug that small tests cannot see

The repaired program, `invcnt_int.cpp`, still has the `int` counter, and it passes all 300 random tests: arrays of 6 elements have at most 15 inversions. Here the stress test's second mode is needed: the largest inputs, compared with an answer we can compute independently. Equation (8.1) gives the exact answer for a reversed array. `gen_invcnt_max.py` writes the reversed array of 200,000 elements and a random one:

```
reversed, invcnt.cpp     : 19999900000
reversed, invcnt_int.cpp : -1474936480
random,   invcnt.cpp     : 10009896365
random,   invcnt_int.cpp : 1419961773
```

The reversed array shows the overflow at once: 19 999 900 000 is what Equation (8.1) says, and a negative count of pairs is impossible. The random array is the more dangerous case: the `int` version prints a positive, plausible number that is simply wrong. Without an independent check you might believe it. For the random array the check is the correct program itself, now trusted because it agrees with the brute force on small inputs and with Equation (8.1) on the extreme one.

8.6 Hunting a Time Limit

8.6.1 Measure, do not guess

A time limit is a statement about the largest input, so a program must be timed on the largest input. **Listing 8.6** is a small stopwatch that runs any command on an input file and reports the wall time and, on Linux and macOS, the peak memory.

Listing 8.6 `measure.py`: runs a command on an input file and reports the time and the peak memory.

```python
# Run a program on an input file and report wall time and peak memory.
# Usage: python3 measure.py INPUT_FILE COMMAND...
# Peak memory needs the resource module (Linux, macOS); on Windows it is skipped.
import subprocess
import sys
import time

try:
    import resource
except ImportError:
    resource = None


def main():
    path, cmd = sys.argv[1], sys.argv[2:]
    with open(path, "rb") as f:
        start = time.perf_counter()
        r = subprocess.run(cmd, stdin=f, stdout=subprocess.DEVNULL)
        elapsed = time.perf_counter() - start
    line = "%-28s %7.3f s" % (" ".join(cmd)[-28:], elapsed)
    if resource is not None:
        peak = resource.getrusage(resource.RUSAGE_CHILDREN).ru_maxrss
        if sys.platform == "darwin":
            peak //= 1024                    # macOS reports bytes, Linux KiB
        line += "  %8.1f MiB" % (peak / 1024)
    if r.returncode != 0:
        line += "  exit code %d" % r.returncode
    print(line)


if __name__ == "__main__":
    main()
```

The peak memory comes from `resource.getrusage`, which reports the largest resident size of the finished child processes; the module does not exist on Windows, and there the column is simply left out. Always compile the program with the judge's flags (Chapter 5) before timing it.

8.6.2 How does the time grow?

A single measurement says whether a program is fast on one input. Several measurements on growing inputs say *how* its time grows, and therefore how it will do on inputs you cannot run. `timing_invcnt.py` runs both INVCNT programs on random arrays of growing size, three times each, and keeps the fastest run. **Table 8.2** and **Figure 8.1** show the result.

Table 8.2 Time of the two INVCNT programs on random arrays of growing size (fastest of three runs, September 2026).

n	Brute force (s)	Merge sort (s)
1000	0.005	0.004
3000	0.026	0.005
10000	0.135	0.003
30000	1.182	0.006
100000	13.406	0.017
200000	53.467	0.033

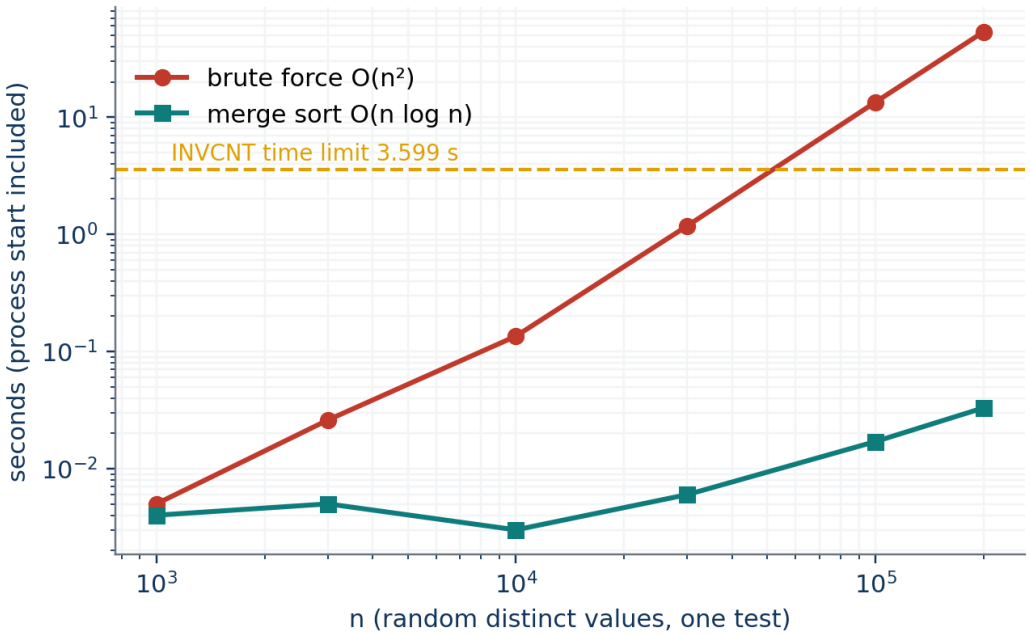


Figure 8.1 The brute force and the merge-sort program on random arrays, on logarithmic scales; the dashed line is INVCNT's time limit.

The numbers for small n are dominated by starting the process, a few milliseconds that have nothing to do with the algorithm. From 10,000 on, the brute force shows its nature. If a program's time grows like n^k, multiplying n by a factor r multiplies the time by r^k, so the exponent can be estimated from two measurements:

$$k \approx \frac{\log(t(n_2)/t(n_1))}{\log(n_2/n_1)} \tag{8.4}$$

From 30,000 to 100,000, $k \approx \log(13.406/1.182)/\log(100\,000/30\,000) = \log 11.34/\log 3.33 \approx 2.02$: quadratic, as the two nested loops promise. The estimate predicts about $13.4 \times 2^2 \approx 54$ seconds for $n = 200\,000$, and the measurement gave 53.467 seconds, fifteen times INVCNT's limit. The merge-sort program grows so slowly that process start still dominates at 100,000; at 200,000 it needs 0.033 seconds. On a logarithmic plot like **Figure 8.1**, a power law is a straight line whose slope is k, which makes the difference visible at a glance.

The practical use of Equation (8.4) is prediction. When a program takes 0.3 seconds at a tenth of the maximum size and grows quadratically, it will take about 30 seconds at the maximum: you know the verdict before submitting.

8.6.3 Python on the largest input

The Python program of **Listing 8.5** took 0.33 seconds on one random array of 200,000 elements. INVCNT does not state the number of test cases; if the judge's file holds several arrays of the maximum size, Python will not pass within 3.6 seconds, while the C++ program needs 0.033 seconds per array. This is the typical position of Python on SPOJ problems with n in the hundreds of thousands and $\Theta(n \log n)$ work written as Python loops: correct, and at the mercy of the number of test cases.

8.7 PALIN: A Wrong Answer and a Hidden Quadratic Step

8.7.1 The problem

SPOJ problem PALIN, "The Next Palindrome" [5], gives t positive integers K of up to 1,000,000 digits, without leading zeros, and asks for each the smallest *palindrome* strictly larger than K: a number that reads the same forward and backward, such as 7, 11, 818 or 12321. The problem page gives the time limit as a range from 2 to 9 seconds, and warns that the input and output are large.

A million digits rules out arithmetic on numbers: the input is a string, and the answer must be built as a string in linear time.

8.7.2 The idea

Let K have n digits. Two cases are special or easy.

- **All nines.** If $K = 99\ldots9$, no palindrome with n digits is larger, and the answer is $10\ldots01$ with $n + 1$ digits.
- **Otherwise** the answer has n digits. A palindrome of n digits is determined by its *left half*, its first $\lceil n/2 \rceil$ digits, and two palindromes of the same length compare exactly as their left halves do.

Let M be the palindrome obtained by *mirroring* K's left half onto its right half. Any palindrome with a smaller left half is smaller than K (it is already smaller in the left half). So if $M > K$, then M is the answer. If $M \leq K$, the answer must have a larger left half, and the smallest larger left half is the left half plus 1, mirrored again. Adding 1 to the left half works like adding 1 to a number: start at the middle digit; while it is 9, set it to 0 and move one digit to the left; then increase that digit by 1. The left half cannot consist only of nines here (then M would be all nines, and $M \leq K$ would force K to be all nines too), so the carry always stops.

Because both strings have n digits, comparing M and K as strings, character by character, compares them as numbers. Every step touches each digit a constant number of times, so the whole algorithm is linear in n.

Two worked examples. For $K = 12921$ the left half is 129, and mirroring gives $M = 12921 = K$, not larger. Increase the left half: the middle digit 9 becomes 0 with a carry, 2 becomes 3, giving 130, and mirroring gives 13031. For $K = 808$, $M = 808 = K$; the middle 0 becomes 1, and the answer is 818.

8.7.3 The programs

Listing 8.7 `palin.cpp`: the next palindrome in linear time on the digit string.

```cpp
// PALIN (SPOJ): the smallest palindrome strictly larger than K,
// where K has up to 1,000,000 digits. Linear time on the digit string.
#include <iostream>
#include <string>

std::string nextPalindrome(std::string s) {
    int n = (int)s.size();
    if (s.find_first_not_of('9') == std::string::npos)   // 9...9 -> 10...01
        return "1" + std::string(n - 1, '0') + "1";
    std::string m = s;
    for (int i = 0; i < n / 2; i++) m[n - 1 - i] = m[i];  // mirror left onto right
    if (m > s) return m;                // same length: string order = number order
    int i = (n - 1) / 2;                // grow the left half by one, from the middle
    while (m[i] == '9') { m[i] = '0'; i--; }             // carry moves left
    m[i]++;
    for (int k = 0; k < n / 2; k++) m[n - 1 - k] = m[k];  // mirror again
    return m;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int t;
    if (!(std::cin >> t)) return 0;
    std::string k;
    while (t-- && std::cin >> k) std::cout << nextPalindrome(k) << '\n';
    return 0;
}
```

Listing 8.7 follows the idea line by line. The Python version, **Listing 8.8**, has one more reason to stay with strings: since Python 3.11, `int()` refuses to convert a string of more than 4300 digits (Chapter 5), so converting K to an integer would not only be slow but would raise an error on the judge's large tests. The helper `bump` adds 1 to a digit string by replacing its trailing nines with zeros and increasing the digit before them.

Listing 8.8 `palin.py`: the next palindrome in Python, on strings only.

```
# PALIN (SPOJ) in Python, on strings: int() refuses to convert strings of
# more than 4300 digits (Python 3.11 and newer), and would be slow anyway.
import sys


def next_palindrome(s):
    n = len(s)
    if s.count("9") == n:                                # 9...9 -> 10...01
        return "1" + "0" * (n - 1) + "1"
    left = s[:(n + 1) // 2]
    m = left + left[:n // 2][::-1]                       # mirror left onto right
    if m > s:
        return m
    left = bump(left)
    return left + left[:n // 2][::-1]


def bump(digits):                    # add 1 to a digit string that is not all 9s
    body = digits.rstrip("9")
    nines = len(digits) - len(body)
    return body[:-1] + chr(ord(body[-1]) + 1) + "0" * nines


def main():
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    out = [next_palindrome(x.decode()) for x in data[1:1 + t]]
    sys.stdout.write("\n".join(out) + "\n")


if __name__ == "__main__":
    main()
```

Our sample contains the two worked examples and three small cases: 7 (the next palindrome is 8), 9 (all nines: 11) and 199 (mirroring gives 191, which is smaller, and increasing the left half 19 gives 20 and the answer 202).

```
5
7
9
808
12921
199
```

```
8
11
818
13031
202
```

The brute force, **Listing 8.9**, does what the statement says with no idea at all: count upward from $K + 1$ until the number reads the same backward. It is hopeless for a million digits and instant for five.

Listing 8.9 `brute_palin.py`: the PALIN brute force counts upward until it meets a palindrome.

```
# Brute force for PALIN: count upwards until a palindrome appears.
import sys

data = sys.stdin.read().split()
for x in data[1:1 + int(data[0])]:
    k = int(x) + 1
    while str(k) != str(k)[::-1]:
        k += 1
    print(k)
```

The generator `gen_palin.py` writes numbers of 1 to 5 digits, drawing each digit from a pool with extra nines and zeros, and turns 30% of them into palindromes, because the interesting cases are exactly "K is already a palindrome" and "the carry runs through several nines".

8.7.4 A wrong answer: the missing carry

`palin_bug.cpp` differs from **Listing 8.7** in one line: it increases the middle digit without the carry loop, as if the middle digit could never be 9.

```
<     while (m[i] == '9') { m[i] = '0'; i--; }            // carry moves left
<     m[i]++;
---
>     m[i]++;                             // slip: no carry when m[i] is '9'
```

On the sample it prints `8 11 818 12:21 1:1`, and the stress test finds it on the first seed:

```
python3 stress.py "python3 gen_palin.py" "python3 brute_palin.py" build/palin_bug
```

```
MISMATCH on seed 1
input:
2
88999
2289
expected: 89098 2332
received: 88:88 2332
```

The character after '9' in the character table is ':', so "increase the digit 9" produced a colon. The same slip in a language with numeric digits would have produced a 10 in one position, or a silently wrong 0; the colon is a lucky, loud symptom. With the carry loop restored, the stress test reports `OK: 300 random tests agree` for both `palin.cpp` and `palin.py`.

8.7.5 A time limit that random tests cannot see

The third variant, `palin_slow.cpp`, is correct: it passes every stress test. It differs in how it builds the mirrored right half after increasing the left half: instead of copying digits into place, it builds a new string by inserting each digit at the *front*.

```
<     for (int k = 0; k < n / 2; k++) m[n - 1 - k] = m[k];  // mirror again
<     return m;
---
>     std::string left = m.substr(0, (n + 1) / 2), right;
>     for (int k = 0; k < n / 2; k++) right.insert(0, 1, left[k]);  // slow
>     return left + right;
```

Each insertion at the front of a string shifts every character already there one place to the right, so building a right half of h characters costs $1 + 2 + \cdots + h \approx h^2/2$ character moves. For $n = 10^6$, $h = 500\,000$ and that is about 1.25×10^{11} moves: a hidden quadratic step inside a linear algorithm.

We timed the programs on two inputs of a million digits. The first is random; the second, `7` followed by 499,999 threes and 500,000 eights, is built so that mirroring gives a smaller number and the slow path must run. **Table 8.3** shows the results.

Table 8.3 Time of three PALIN programs on inputs of a million digits.

Input (10^6 digits)	palin.cpp	palin.py	palin_slow.cpp
random digits	0.004 s	0.021 s	0.011 s
7, then 3s, then 8s (forces the increase)	0.005 s	0.014 s	2.471 s

On the random input the slow program is as fast as the others, because for most random numbers the mirror is already larger and the slow branch is never reached. Only the input *designed* to force

the increase exposes it. The lesson matters as much as the stress test itself: **the worst case must be constructed, not hoped for.** For every candidate, ask which input makes each loop run longest, and write a generator for that input. For PALIN it is "mirroring gives a smaller number"; for a sort it is often "already sorted" or "all equal"; for a hash table, "all keys collide".

The measurement also shows the quadratic growth: on the corresponding input of 100,000 digits the same slow program took 0.011 seconds, most of it spent starting the process, so ten times more digits cost more than two hundred times more time; with h^2 steps the algorithmic part alone grows a hundredfold.

8.8 More Tools for Runtime Errors

8.8.1 Stop at the line that goes wrong

A runtime error on the judge, or a wrong answer that depends on garbage in memory, usually comes from reading or writing outside an array. C++ does not check indexes by default, so such a program can run and print a plausible number. **Listing 8.10** is a four-line example: a prefix-sum loop that reads `a[i]` where it should read `a[i - 1]`, and so reads one element past the end in its last step.

Listing 8.10 `bounds_demo.cpp`: a prefix-sum loop that reads one element past the end of the vector.

```cpp
// A prefix-sum loop that reads one element past the end of a vector.
// Plain -O2 prints a number; -D_GLIBCXX_DEBUG or -fsanitize=address stop
// the program at the exact line instead.
#include <cstdio>
#include <vector>

int main() {
    std::vector<int> a = {4, 1, 3, 2, 5};
    std::vector<long long> prefix(a.size() + 1, 0);
    for (std::size_t i = 1; i <= a.size(); i++)
        prefix[i] = prefix[i - 1] + a[i];   // slip: should be a[i - 1]
    std::printf("sum of all = %lld\n", prefix[a.size()]);
    return 0;
}
```

Compiled normally with `-O2`, it prints a wrong sum and no warning; the sum of 4, 1, 3, 2, 5 is 15.

```
sum of all = 11
```

Compiled with `-D_GLIBCXX_DEBUG`, g++'s standard library checks every index [6], and the program stops at the first bad access with a message that names the problem (the report is shortened here):

```
/usr/include/c++/13/debug/vector:442:
Error: attempt to subscript container with out-of-bounds index 5, but
container only holds 5 elements.
```

Compiled with `-fsanitize=address` and `-g`, the address sanitizer [7] reports the exact source line:

```
ERROR: AddressSanitizer: heap-buffer-overflow on address 0x503000000054
READ of size 4 at 0x503000000054 thread T0
    #0 0x562df89a2a2f in main bounds_demo.cpp:11
```

Line 11 is the loop body. Either tool turns an invisible bug into a stopped program with an address; run your stress tests once with each, as Chapter 5 recommended.

8.8.2 Assertions

An assertion states something that must be true at a point of the program: `assert(0 <= i && i < n);` or `assert(total % n == 0);`. If it is false, the program stops and prints the file, the line and the condition. Assertions cost almost nothing, document what you believe, and turn a silent wrong answer into a loud runtime error during stress testing. They are active unless the program is compiled with `-DNDEBUG`, so leave them in during testing; on the judge a failed assertion is a runtime error, which tells you more than a wrong answer would.

8.8.3 Print to the error stream

When you add printing to understand a failing input, print to `stderr` (`std::cerr` or `fprintf(stderr, ...)` in C++, `print(..., file=sys.stderr)` in Python). The judge and the harness compare only standard output, so debugging lines on `stderr` do not change the verdict if you forget one, although they do cost time on large inputs.

8.8.4 Bisecting a file of many test cases

Sometimes you do have a failing file, a large one with hundreds of test cases, for instance from a friend's generator or from a test published after a contest, and the program's output differs from the expected output somewhere. If the bug is a leak between test cases (Chapter 7), every case after some point may be wrong, and the first wrong case is the interesting one: it is where the leaked state first mattered. Checking the cases one by one works but is slow when each check needs a manual look.

The failure has a useful property: once a leak has corrupted the state, it stays corrupted. So the predicate "the first k test cases are all answered correctly" is true up to some k^* and false after it, and k^* can be found by *binary search*: run the program on the first k cases for k halfway through the unknown range, compare, and keep the half that contains the change. With 500 test cases, about $\log_2 500 \approx 9$ runs find the first failing case, instead of up to 500. The same halving works on the *input* of one large test case whenever the failure persists once it appears, which is Bu Endang's method of Section 8.1 applied to a file. It is also the idea behind `git bisect`, which finds the commit that introduced a bug by halving the history.

Bisection needs the failure to be monotone. If test case 17 fails because of its own content and test case 18 passes, the predicate is not monotone, and a linear scan, or simply comparing the outputs line by line with `diff`, is the right tool.

8.8.5 When the answer is not unique

Some problems accept any of several correct answers: "print any shortest path", "print any valid arrangement". Comparing with a brute force's output then fails even for correct programs. The fix is a *checker*: a program that reads the input and the candidate's output and verifies the answer's properties, for example that the printed path uses real edges, starts and ends at the right vertices, and has the length the brute force found. The harness stays the same; only the comparison is replaced by a call to the checker.

8.9 Using It for Real

The routine of this chapter fits on an index card.

On a wrong answer:

1. Write the brute force from the statement, without cleverness, and a generator for small valid inputs aimed at the dangerous shapes.
2. Run the stress test. When it fails, shrink the input and trace the smallest case by hand.
3. When it passes, add the largest inputs and compare with an independent answer: a formula, a second fast method, or the extreme case whose answer you know.

On a time limit:

1. Build the worst input the constraints allow, not only a random one; ask which input makes each loop run longest.
2. Time the program with the judge's flags on growing sizes and estimate the exponent with Equation (8.4).
3. If the exponent is too large, change the algorithm; if it is right but the constant is too large, look for hidden quadratic steps (insertions at the front, `erase` in a loop, string concatenation in a loop, clearing large arrays per test case) before reaching for the flags of Chapter 5.

On a runtime error: compile with `-D_GLIBCXX_DEBUG` and with `-fsanitize=address,undefined` and rerun the stress tests; check recursion depth against Chapter 6.

A note on time. The whole routine sounds long, but in practice the brute force of a typical SPOJ problem is ten lines, the generator another ten, and the harness is written once for the rest of your life. Fifteen minutes spent on them usually saves more than fifteen minutes of staring at code, and it saves something more valuable in a contest: the confidence to submit, or the certainty that you should not yet. With experience, you will start writing the brute force *first*, before the fast solution, because it also checks your understanding of the statement on the samples.

Keep the three programs next to your solution, in a `tests` folder like the ones in this book, and they cost almost nothing to reuse. The same habit works far from contests. Bu Endang's two counts are a stress test of her notebook; a cooperative that moves its accounts from a paper ledger to a spreadsheet can keep both for a few months and compare the monthly totals, shrinking any difference to the week, the day and the entry; and anyone who writes a formula in a spreadsheet can check it against a hand calculation on three or four simple rows before trusting it on three thousand.

8.10 Common Mistakes, and How to Avoid Them

- **A brute force that shares the candidate's idea.** If both use the same formula, they agree on its mistakes. The brute force should follow the statement, not the solution.
- **A generator that writes invalid inputs.** A difference on an input the problem forbids is not a bug. Check the generator against the constraints: distinct values in INVCNT, no leading zeros in PALIN, no trailing zeros in ADDREV.
- **Only small random tests.** They cannot see overflow or slowness. Add the largest inputs and the constructed worst cases.
- **Timing an unoptimized build, or a random input only.** Time with `-O2` on the worst input.
- **Not saving the seed.** A failure you cannot reproduce is a failure you cannot fix. Generators must be deterministic for a given seed.

- **Debugging the big failing input.** Shrink first. A two-element failure explains itself; a two-hundred-element one rarely does.
- **Stopping at the first fix.** After repairing a bug, rerun the whole stress test, then the largest inputs: the first fix may uncover the next bug, as `invcnt_int.cpp` did.

8.11 Chapter Summary

- The judge shows only a verdict, so bugs must be found on your own machine with a second method: a brute force that is obviously right, compared with the candidate on many generated inputs (**Listing 8.1**).
- Most bugs appear on small inputs (the small scope hypothesis); shrinking a failing input to its smallest form makes the cause visible, as the two-element array exposed the off-by-one in Equation (8.2).
- Bugs of size need other inputs: the largest arrays against an independent answer such as Equation (8.1) catch overflow; constructed worst cases catch hidden slowness that random inputs miss (**Table 8.3**).
- Timing on growing inputs and Equation (8.4) reveal how time grows and predict the verdict on sizes you cannot run; INVCNT's brute force is quadratic, the merge sort $\Theta(n\log n)$ (Equation (8.3)).
- `-D_GLIBCXX_DEBUG`, the address sanitizer and assertions stop a program at the line where it goes wrong.
- INVCNT is solved by counting cross inversions while merging; PALIN by mirroring the left half and, if necessary, increasing it with a carry, in linear time on the digit string.

8.12 Exercises

1. Count the inversions of 5,1,4,2,3 by hand, first by listing the pairs and then by following the merges of **Listing 8.4** and adding Equation (8.2) at each step. Check with the program.
2. The file `invcnt_bug.cpp` has two slips. Suppose only the `int` slip were present. What is the smallest n for which a reversed array makes the program print a wrong answer? Give the printed value.
3. Write the smallest-possible failing input for `palin_bug.cpp`, and explain why the generator's pool of digits "0999912345678" finds such inputs faster than uniform digits would.
4. A candidate for PALIN gives correct answers but, for inputs with an even number of digits, mirrors the left half with an index off by one. Which of the sample's five numbers would expose the slip, and which would not?
5. Write a generator `gen_palin_worst.py` that writes a single number of n digits for which `palin_slow.cpp` must take its slow path, for any $n \geq 2$. Explain why your number forces the increase.
6. Change `stress.py` so that, after finding a failing input for a problem with one array per test case, it tries deleting elements one at a time and keeps each deletion that still fails. What does the procedure guarantee about the input it ends with?

8.13 Solutions to the Exercises

1. The pairs in the wrong order are (5, 1), (5, 4), (5, 2), (5, 3), (4, 2), (4, 3): 6 inversions. With the merges: the array is split into [5,1,4] and [2,3] (lo = 0, mid = 2, hi = 4). The left part splits into [5,1]

and [4]; [5,1] merges into [1,5] adding $0 - 0 + 1 = 1$ when 1 is taken. Merging [1,5] with [4]: 1 is taken from the left, then 4 is smaller than 5, adding $1 - 1 + 1 = 1$; running total 2. [2,3] adds nothing. The final merge of [1,4,5] with [2,3]: 1 from the left; 2 is smaller than 4, adding $2 - 1 + 1 = 2$; 3 is smaller than 4, adding 2 again; total $2 + 2 + 2 = 6$. The program prints 6.

2. The reversed array has $n(n-1)/2$ inversions by Equation (8.1); the `int` overflows when this exceeds $2^{31} - 1 = 2\,147\,483\,647$. $n(n-1)/2 > 2\,147\,483\,647$ first holds for $n = 65\,537$, where the count is 2 147 516 416; the wrapped value is $2\,147\,516\,416 - 2^{32} = -2\,147\,450\,880$. (Strictly, the overflow is undefined behavior; this is the value it produces on common compilers.)

3. $K = 9$ is all nines and handled separately, and one-digit numbers below 9 never carry. The smallest failing input is $K = 191$: mirroring gives 191, not larger; the middle digit 9 is increased without carry to ':', and the program prints `1:1` instead of 202. (99 is all nines; for two digits, $K = 19$ mirrors to 11 and increases the 1 to 2, giving 22, correctly.) The pool contains the digit 9 four times out of thirteen, so a random middle digit is 9 with probability about 0.31 instead of 0.1, and the failing shapes appear three times as often.

4. Only the number with an even number of digits can expose it: 12921, 808, 199, 7 and 9 have odd lengths, so none of the five does. This is why the generator draws lengths from 1 to 5: half of the random numbers have an even length.

5. Write `7`, then $\lceil n/2 \rceil - 1$ threes, then $\lfloor n/2 \rfloor$ eights. The left half is 733 … 3; mirroring puts 3 … 37 into the right half, whose first digit 3 is smaller than the 8 in the same position of K (for $n = 2$ the left half is just 7, the mirror is 77 and K is 78). So $M < K$, the increase is forced, and the slow insertion runs over the whole right half. Its first digit 7 guarantees no leading zero and that the number is not all nines.

6. After the stress test fails, loop: for each position p of the array, form the input without element p (adjusting n), run the brute force and the candidate, and if they still disagree, keep the smaller input and start over; stop when no single deletion fails. The result is *1-minimal*: removing any one element makes the failure disappear. It is not necessarily the smallest failing input overall, but it is usually small enough to trace by hand, and the procedure needs at most n^2 runs.

References

[1] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, rev. ed. Cambridge, MA, USA: MIT Press, 2012.

[2] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 2nd ed. Cham, Switzerland: Springer, 2020.

[3] Sphere Online Judge, "INVCNT - Inversion Count." [Online]. Available: https://www.spoj.com/problems/INVCNT/ (accessed Sep. 2026).

[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] Sphere Online Judge, "PALIN - The Next Palindrome." [Online]. Available: https://www.spoj.com/problems/PALIN/ (accessed Sep. 2026).

[6] Free Software Foundation, "Debug Mode: Using," *The GNU C++ Library Manual*. [Online]. Available: https://gcc.gnu.org/onlinedocs/libstdc++/manual/debug_mode_using.html (accessed Sep. 2026).

[7] Free Software Foundation, "Program Instrumentation Options," *Using the GNU Compiler Collection (GCC)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html (accessed Sep. 2026).

Chapter 9

Dynamic Programming Patterns

Many SPOJ problems look impossible at first sight: there are more ways to choose, split, walk or combine than any computer could ever try. Yet a large family of them falls to one idea, *dynamic programming*: describe the answer for a big input in terms of answers for smaller inputs, compute each small answer once, and keep it in a table. By the end of this chapter you will be able to find the *state* of a problem, write its recurrence with base cases, choose the order in which to fill the table, and turn all of that into short, fast programs in C++17 and Python 3. You will work through six real SPOJ problems, each one a pattern that returns again and again: a line of choices (FARIDA), counting ways (ACODE), paths through a grid (BYTESM2), items under a budget (KNAPSACK), two strings (EDIST) and intervals (MIXTURES). Every program is checked against a brute force on hundreds of random inputs and timed on the largest inputs the problems allow.

9.1 Bu Wati's Market Days

Bu Wati sells fruit at the weekly markets around her village: rambutan and mangosteen in the rainy months, mango and durian when their seasons come. There is a market somewhere nearby every day, and each one is different. The Monday market by the bus terminal is quiet; the Thursday market near the factory is busy when the workers have just been paid; the Saturday market draws families from the city. From her notebook she knows roughly how much she can earn at each of the next fourteen markets.

There is one catch. After a market day she needs a full day at home to pick, sort and pack fresh fruit. So she can never sell on two days in a row. Which market days should she choose to earn the most over the next two weeks?

The obvious ideas fail. "Take the best day, then the best day that is still allowed, and so on" sounds sensible, but with earnings 5, 6 and 5 on three consecutive days, taking the 6 blocks both 5s: 6 is less than 10. "Take every second day" fails on 1, 9, 1, 1, 9, where the best plan takes the two 9s, which are three days apart. Trying every plan works but does not scale: fourteen days already allow 987 valid plans, and a year of daily markets allows about 10^{76} of them.

Bu Wati's question is, word for word, the SPOJ problem FARIDA (Princess Farida) [1]: a row of monsters, each guarding some coins, and a rule that you may not take coins from two neighboring monsters. The same shape appears in rostering a single worker who needs rest days, in choosing non-adjacent plots to plant in one season, and in hundreds of other everyday choices. This chapter solves it, and five of its cousins, with one way of thinking.

9.2 The Idea in Plain Words

9.2.1 Smaller questions with the same shape

Look at the last market day, day n. Either Bu Wati sells on it or she does not.

- If she does **not** sell on day n, her best plan is simply her best plan for days 1 to $n - 1$.

- If she **does** sell on day n, she earns c_n that day, cannot sell on day $n-1$, and for days 1 to $n-2$ she should again use the best plan there is.

Both cases ask the same question as the original, only about a shorter row of days. Write $best(i)$ for the most she can earn using only days 1 to i. Then

$$best(i) = \max(best(i-1),\ best(i-2) + c_i), \qquad best(0) = 0,\ best(1) = c_1 \tag{9.1}$$

Equation (9.1) is the whole solution. It is called a *recurrence*: it defines each value by values with smaller arguments, and the two *base cases* stop the chain. Nothing about "greedy" choices is needed; the recurrence simply compares the two possibilities for the last day, and each possibility is answered exactly by a smaller instance of the same question.

> **DEFINITION.** A *state* is a short description of a subproblem, such as "the first i days". The *value* of a state is the best answer (or the number of ways, or the least cost) for that subproblem. A *recurrence* expresses the value of a state through the values of smaller states; *base cases* give the values of the smallest states directly. *Dynamic programming* (DP) computes the value of every state that is needed exactly once, stores it, and reads it back whenever it is needed again [2], [3].

9.2.2 Working it by hand

Take five market days with earnings $c = (4,1,1,9,2)$. Fill in $best(i)$ from left to right by Equation (9.1):

- $best(0) = 0$ and $best(1) = 4$.
- $best(2) = \max(4,\ 0+1) = 4$: skipping day 2 is better.
- $best(3) = \max(4,\ 4+1) = 5$: sell on days 1 and 3.
- $best(4) = \max(5,\ 4+9) = 13$: sell on day 4, plus the best of days 1 and 2.
- $best(5) = \max(13,\ 5+2) = 13$: skipping day 5 is better.

The answer is 13: days 1 and 4. Each line used only the two lines above it, so the work is one short step per day. For n days that is $O(n)$ time, instead of the exponentially many plans that brute force would try.

9.2.3 Five questions for every DP

Every DP solution in this chapter, and almost every one you will write, answers the same five questions. Asking them in this order is the most reliable way to go from a problem statement to a correct program.

1. **State.** What is a subproblem, and which few numbers describe it? Here: the prefix length i.
2. **Recurrence.** How does the value of a state follow from smaller states? Think about the *last decision*: here, whether to use day i.
3. **Base cases.** Which states can be answered without the recurrence? Here: $i = 0$ and $i = 1$.
4. **Order.** In which order must the states be filled so that every value is ready before it is read? Here: increasing i.
5. **Answer.** Which state, or which combination of states, is the answer? Here: $best(n)$.

The running time of a DP is (number of states) × (work per state). Here that is $n \times O(1)$. Chapter 3 taught you to read the constraints first; with DP you estimate the number of states and the work

per state and compare the product with the roughly 10^8 simple steps per second a judge gives you.

> **KEY INSIGHT:** DP is not a clever formula. It is the discipline of naming the subproblems precisely. Once the state is right, the recurrence usually writes itself by asking "what was the last decision?", and the program is a loop that fills a table in the right order.

9.3 A First Attempt: Plain Recursion

The most direct program for Equation (9.1) is a recursive function that calls itself for $i-1$ and $i-2$. It is correct, and it is a good first check, but it is slow for a reason worth seeing with your own eyes. **Listing 9.1** evaluates the same recurrence in three ways: plain recursion, recursion with a *memo* (a table of already computed values, where -1 means "not yet computed"), and a table filled from left to right. It counts the function calls.

Listing 9.1 `dp_intro.cpp`: plain recursion, recursion with a memo, and a table, all computing Equation (9.1).

```cpp
// Three ways to evaluate best(i) = max(best(i-1), best(i-2) + c[i]):
// plain recursion, recursion with a memo, and a table filled left to right.
#include <algorithm>
#include <cstdio>
#include <iostream>
#include <vector>
using namespace std;

vector<long long> c;
long long calls = 0;

long long plain(int i) {  // best total over monsters 0..i-1
    calls++;
    if (i <= 0) return 0;
    if (i == 1) return c[0];
    return max(plain(i - 1), plain(i - 2) + c[i - 1]);
}

vector<long long> memo;  // -1 means "not computed yet"; real answers are >= 0

long long memoized(int i) {
    calls++;
    if (i <= 0) return 0;
    if (i == 1) return c[0];
    if (memo[i] != -1) return memo[i];
    return memo[i] = max(memoized(i - 1), memoized(i - 2) + c[i - 1]);
}

long long table(int n) {
    vector<long long> best(n + 1, 0);
    if (n >= 1) best[1] = c[0];
    for (int i = 2; i <= n; i++) best[i] = max(best[i - 1], best[i - 2] + c[i - 1]);
    return best[n];
}

int main() {
    cout << " n    plain answer      plain calls  memo calls  table answer\n";
    for (int n : {10, 20, 30, 40}) {
        c.assign(n, 0);
        for (int i = 0; i < n; i++) c[i] = (i * 37 + 11) % 50;  // fixed coins
        calls = 0;
        long long a = plain(n);
        long long plainCalls = calls;
        memo.assign(n + 1, -1);
        calls = 0;
        memoized(n);
```

Listing 9.1 (Continued) *dp_intro.cpp: plain recursion, recursion with a memo, and a table, all computing Equation (9.1).*

```cpp
        printf("%2d %15lld %16lld %11lld %13lld\n", n, a, plainCalls, calls,
               table(n));
    }
    return 0;
}
```

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o dp_intro dp_intro.cpp
$ ./dp_intro
```

```
 n    plain answer      plain calls  memo calls  table answer
10             180              177          19           180
20             310            21891          39           310
30             467          2692537          59           467
40             612        331160281          79           612
```

All three give the same answers, but look at the call counts. Plain recursion made 177 calls for 10 days and more than 331 million for 40 days: each extra day multiplies the work by about 1.6. The reason is that $plain(i)$ asks for $plain(i-1)$ and $plain(i-2)$, and $plain(i-1)$ asks for $plain(i-2)$ again, so the same small questions are asked over and over. The number of calls grows like the Fibonacci numbers, roughly 1.618^n. With a memo, each state is computed once and later calls return at once: $2n-1$ calls. The table version has no calls at all, just a loop.

These are the two standard ways to write a DP:

- **Top-down (memoization):** keep the recursive function and add a memo. It is close to the recurrence and computes only the states that are actually reached. Its costs are the function-call overhead and the recursion depth: with $n = 10^5$, a recursive C++ function may overflow the stack, and Python stops at a recursion limit of 1000 by default.
- **Bottom-up (tabulation):** fill the table in an order where every value is ready before it is needed. It is faster, has no depth problem, and makes memory savings easy to see.

In a contest, write bottom-up whenever the order is easy to see, and keep top-down for states whose order is awkward (many DP over trees or over subsets start that way). Keep the memo sentinel safe: -1 works here because every real answer is at least 0; for a problem whose answers can be negative, use a separate `bool` table of "computed" flags.

9.4 Pattern 1: A Line of Choices (FARIDA)

The problem in our words. FARIDA [1] gives several test cases. Each has N monsters in a row, $0 \le N \le 10^4$, and monster i guards between 0 and 10^9 coins. Taking coins from a monster is allowed only if you have not taken from the monster right before it. Print the most coins you can collect, in the form `Case k: x`.

The idea is Equation (9.1). Since $best(i)$ needs only $best(i-1)$ and $best(i-2)$, the program keeps just two numbers instead of a whole table. The variable `before` holds $best(i-2)$ and `last` holds $best(i-1)$; after each monster they move one place to the right.

The pitfalls. Three details decide between Accepted and Wrong Answer here. First, N may be 0; a program that reads the first coin before the loop, or sets $best(1)$ from an array element that does not exist, fails or crashes on that case. The rolling version below handles it without a special test, because with no monsters `last` stays 0. Second, the total can reach $10^4 \times 10^9 = 10^{13}$, far beyond the $2^{31}-1 \approx 2.1 \times 10^9$ of `int`: use `long long` (Chapter 6). Third, the output format: a capital C, a space, the case number, a colon and a space. **Listing 9.2** is the C++ program and **Listing 9.3** the Python one.

Listing 9.2 `farida.cpp`: FARIDA with two rolling values.

```cpp
// FARIDA (Princess Farida): the most coins from a line of monsters,
// never taking from two neighbors. Linear DP with two rolling values.
#include <algorithm>
#include <iostream>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;
    for (int tc = 1; tc <= t; tc++) {
        int n;
        cin >> n;
        long long before = 0;  // best[i-2]: best total over the first i-2 monsters
        long long last = 0;    // best[i-1]: best total over the first i-1 monsters
        for (int i = 0; i < n; i++) {
            long long c;
            cin >> c;
            long long now = max(last, before + c);  // skip monster i, or take it
            before = last;
            last = now;
        }
        cout << "Case " << tc << ": " << last << '\n';
    }
    return 0;
}
```

Listing 9.3 `farida.py`: the same solution in Python, reading all input at once.

```python
"""FARIDA (Princess Farida): the most coins from a line of monsters,
never taking from two neighbors. Linear DP with two rolling values."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    pos = 1
    out = []
    for tc in range(1, t + 1):
        n = int(data[pos])
        pos += 1
        before = last = 0  # best over the first i-2 and the first i-1 monsters
        for c in map(int, data[pos : pos + n]):
            before, last = last, max(last, before + c)  # skip it, or take it
        pos += n
        out.append(f"Case {tc}: {last}")
    print("\n".join(out))


main()
```

The Python version reads the whole input with `sys.stdin.buffer.read().split()`, the fast pattern from Chapter 4, and updates both rolling values in one tuple assignment, which is exactly the "shift one place to the right" of the C++ version. We run both on two test cases: the five days from Section 9.2.2 and a case with no monsters at all.

```
$ printf '2\n5\n4 1 1 9 2\n0\n' | ./farida
$ printf '2\n5\n4 1 1 9 2\n0\n' | python3 ../python/farida.py
```

```
Case 1: 13
Case 2: 0
Case 1: 13
Case 2: 0
```

Both print 13 for the first case, as computed by hand, and 0 for the empty row. The work is $O(N)$ per test case and the memory is $O(1)$ beyond the input, so both languages are far inside the time limit.

9.5 Pattern 2: Counting Ways (ACODE)

The problem in our words. In ACODE (Alphacode) [4], a message of capital letters has been encoded by writing A as 1, B as 2, and so on up to Z as 26, with no separators. Given the digit string, up to 5000 digits long, count how many letter messages could have produced it. The input holds several strings and ends with a line containing a single 0. The problem promises that every string is a valid encoding and that the count fits in a signed 64-bit integer.

The state. Let $ways(i)$ be the number of ways to read the first i digits. The last letter of such a reading used either the last digit alone (possible when that digit is 1 to 9) or the last two digits together (possible when they form a number from 10 to 26). The two cases never overlap, because they end with letters of different lengths, so their counts add:

$$ways(i) = [\, s_i \neq 0 \,] \cdot ways(i-1) + [\, 10 \le \overline{s_{i-1}s_i} \le 26 \,] \cdot ways(i-2), \qquad ways(0) = 1 \tag{9.2}$$

Here $[\ldots]$ is 1 when the condition inside is true and 0 when it is false, and $\overline{s_{i-1}s_i}$ is the two-digit number formed by the digits s_{i-1} and s_i. The base case $ways(0) = 1$ often surprises beginners: there is exactly one way to read nothing, the empty reading, and without it every count would be 0.

By hand, for the string 25114:

- $ways(0) = 1$; $ways(1) = 1$ (the reading “2”).
- $ways(2) = 1 + 1 = 2$: “2 5” and “25”.
- $ways(3) = 2 + 0 = 2$: the digit 1 alone; “51” is not a letter.
- $ways(4) = 2 + 2 = 4$: the digit 1 alone, or “11”.
- $ways(5) = 4 + 2 = 6$: the digit 4 alone, or “14”.

Pitfalls. The digit 0 has no letter of its own, so it can only be the second digit of 10 or 20; the factor $[s_i \neq 0]$ takes care of that. A pair such as “05” or “30” is not a letter, so the two-digit test must check the range 10 to 26, not just “at most 26”. And counting DPs grow fast: a string of 1s of length n has F_{n+1} readings (the Fibonacci numbers again), which passes 2^{63} at about 90 digits. The problem guarantees that the answer fits, so `long long` is enough in C++; Python’s integers never overflow at all. **Listing 9.4** and **Listing 9.5** turn Equation (9.2) into a loop over the prefixes.

Listing 9.4 `acode.cpp`: counting readings with Equation (9.2).

```cpp
// ACODE (Alphacode): count the ways to read a digit string as letters,
// A=1, B=2, ..., Z=26. Counting DP over prefixes; input ends with "0".
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    string s;
    while (cin >> s && s != "0") {
        int n = s.size();
        vector<long long> ways(n + 1,
                               0);  // ways[i]: readings of the first i digits
        ways[0] = 1;                // the empty prefix has one reading
        for (int i = 1; i <= n; i++) {
            if (s[i - 1] != '0') ways[i] += ways[i - 1];  // one-digit letter 1..9
            if (i >= 2) {
                int two = (s[i - 2] - '0') * 10 + (s[i - 1] - '0');
                if (two >= 10 && two <= 26)
                    ways[i] += ways[i - 2];  // letter 10..26
            }
        }
        cout << ways[n] << '\n';
    }
    return 0;
}
```

Listing 9.5 `acode.py`: the same count in Python.

```python
"""ACODE (Alphacode): count the ways to read a digit string as letters,
A=1, ..., Z=26. Counting DP over prefixes; input ends with "0"."""

import sys


def count(s: str) -> int:
    ways = [0] * (len(s) + 1)  # ways[i]: readings of the first i digits
    ways[0] = 1
    for i in range(1, len(s) + 1):
        if s[i - 1] != "0":
            ways[i] += ways[i - 1]  # one-digit letter 1..9
        if i >= 2 and 10 <= int(s[i - 2 : i]) <= 26:
            ways[i] += ways[i - 2]  # two-digit letter 10..26
    return ways[len(s)]


def main() -> None:
    out = []
    for token in sys.stdin.read().split():
        if token == "0":
            break
        out.append(str(count(token)))
    print("\n".join(out))


main()
```

```
$ printf '25114\n1111111111\n3333333333\n2101\n0\n' | ./acode
$ printf '25114\n1111111111\n3333333333\n2101\n0\n' | python3 ../python/acode.py
```

```
6
89
1
1
6
89
1
1
```

The first four lines come from C++, the next four from Python. The string 25114 has the 6 readings found by hand; ten 1s have $F_{11} = 89$; ten 3s have only one reading, because no two 3s form a letter; and 2101 has exactly one, B J A, because the 0 forces “10”.

9.6 Pattern 3: Paths in a Grid (BYTESM2)

The problem in our words. In BYTESM2 (Philosophers Stone) [5], a floor of h rows and w columns of tiles, each with some number of stones, must be crossed from the top row to the bottom row. From a tile you may step to the tile directly below or to one of the two tiles diagonally below. You may start anywhere in the top row and finish anywhere in the bottom row. Collect as many stones as possible. There are several test cases; h and w are at most 100.

The state is a tile: let $best(r, c)$ be the most stones collected on a walk that ends on the tile in row r, column c. The last step came from one of the three tiles above it:

$$best(r,c) = a_{r,c} + \max(best(r-1,c-1),\ best(r-1,c),\ best(r-1,c+1)) \tag{9.3}$$

with $best(0, c) = a_{0,c}$ in the top row, and with the tiles outside the floor simply left out of the maximum. The answer is the largest value in the bottom row. The order is row by row, and each row needs only the row above, so two rows of memory are enough; **Listing 9.6** and **Listing 9.7** keep exactly two.

By hand, on the floor

```
3 1 7
2 8 1
4 1 6
```

the top row gives (3,1,7). The middle row gives $2 + \max(3,1) = 5$, $8 + \max(3,1,7) = 15$ and $1 + \max(1,7) = 8$. The bottom row gives $4 + \max(5,15) = 19$, $1 + \max(5,15,8) = 16$ and $6 + \max(15,8) = 21$. The answer is 21, from the walk $7 \to 8 \to 6$. Notice that a greedy walk that starts at the largest top tile happens to work here, but it fails as soon as a large tile lies below a small one.

Listing 9.6 `bytesm2.cpp`: grid DP with two rows.

```cpp
// BYTESM2 (Philosophers Stone): walk from the top row to the bottom row,
// each step going straight down or diagonally down; collect the most stones.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;
    while (t--) {
        int h, w;
        cin >> h >> w;
        vector<long long> prev(w, 0), cur(w);  // best totals ending in each column
        for (int r = 0; r < h; r++) {
            for (int col = 0; col < w; col++) {
                long long stones;
                cin >> stones;
                long long from = prev[col];  // the row above; zeros before row 0
                if (col > 0) from = max(from, prev[col - 1]);
                if (col + 1 < w) from = max(from, prev[col + 1]);
                cur[col] = stones + from;
            }
            swap(prev, cur);
        }
        cout << *max_element(prev.begin(), prev.end()) << '\n';
    }
    return 0;
}
```

Listing 9.7 `bytesm2.py`: the same DP in Python, with a padded row instead of border tests.

```python
"""BYTESM2 (Philosophers Stone): top row to bottom row, straight down or
diagonally down at each step; collect the most stones."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    pos = 1
    out = []
    for _ in range(t):
        h, w = int(data[pos]), int(data[pos + 1])
        pos += 2
        prev = [0] * w  # best totals ending in each column of the row above
        for _ in range(h):
            row = list(map(int, data[pos : pos + w]))
            pos += w
            # pad with a very small value so column 0 and column w-1 need no test
            pad = [-1] + prev + [-1]
            prev = [row[c] + max(pad[c], pad[c + 1], pad[c + 2]) for c in range(w)]
        out.append(str(max(prev)))
    print("\n".join(out))


main()
```

The two programs deal with the borders in two different ways. The C++ version tests `col > 0` and `col + 1 < w` before looking left or right. The Python version pads the previous row with -1 on both sides; since every real total is at least 0, the padding can never win a maximum, and the list comprehension needs no tests. Padding with a value that can never be chosen is a trick worth

remembering for every grid DP. In the C++ version the row before the top row is all zeros, which gives the base case $best(0, c) = a_{0,c}$ for free.

```
$ printf '1\n3 3\n3 1 7\n2 8 1\n4 1 6\n' | ./bytesm2
$ printf '1\n3 3\n3 1 7\n2 8 1\n4 1 6\n' | python3 ../python/bytesm2.py
```

```
21
21
```

Both print 21. The work is $O(hw)$, at most 10^4 tiles per test case.

9.7 Pattern 4: Choosing Items Under a Budget (KNAPSACK)

The problem in our words. KNAPSACK (The Knapsack Problem) [6] gives a bag of capacity $S \leq 2000$ and $N \leq 2000$ items, each with a size and a value. Choose items, each at most once, whose sizes add up to at most S and whose values add up to as much as possible. Print that value.

This is the famous *0/1 knapsack*: for every item the decision is 0 (leave it) or 1 (take it). Trying all 2^N subsets is hopeless for $N = 2000$, and greedy rules such as "best value per unit of size first" fail on 0/1 problems, as Chapter 10 will show.

The state needs two numbers: how many items we have considered, and how much capacity we may use. Let $best_i(c)$ be the most value using only the first i items with total size at most c. The last decision is about item i, with size s_i and value v_i:

$$best_i(c) = \max(best_{i-1}(c),\ best_{i-1}(c - s_i) + v_i), \qquad best_0(c) = 0 \tag{9.4}$$

where the second option exists only when $s_i \leq c$. The answer is $best_N(S)$. There are $(N + 1)(S + 1)$ states, about four million, with $O(1)$ work each: fast in C++. A full table of `long long` would take $2001 \times 2001 \times 8$ bytes, about 32 MB, which is acceptable but wasteful, because row i needs only row $i - 1$.

9.7.1 One row, and the direction that matters

The usual trick keeps a single row `best[c]` and updates it in place for each item. Before item i is processed, the row holds $best_{i-1}$; afterward it must hold $best_i$. The update for capacity c reads `best[c - size]`, a cell to the *left*. Whether that cell still holds the old value $best_{i-1}(c - s_i)$ or already the new value $best_i(c - s_i)$ depends only on the direction of the loop, as **Figure 9.1** shows for a single item of size 2 and value 3.

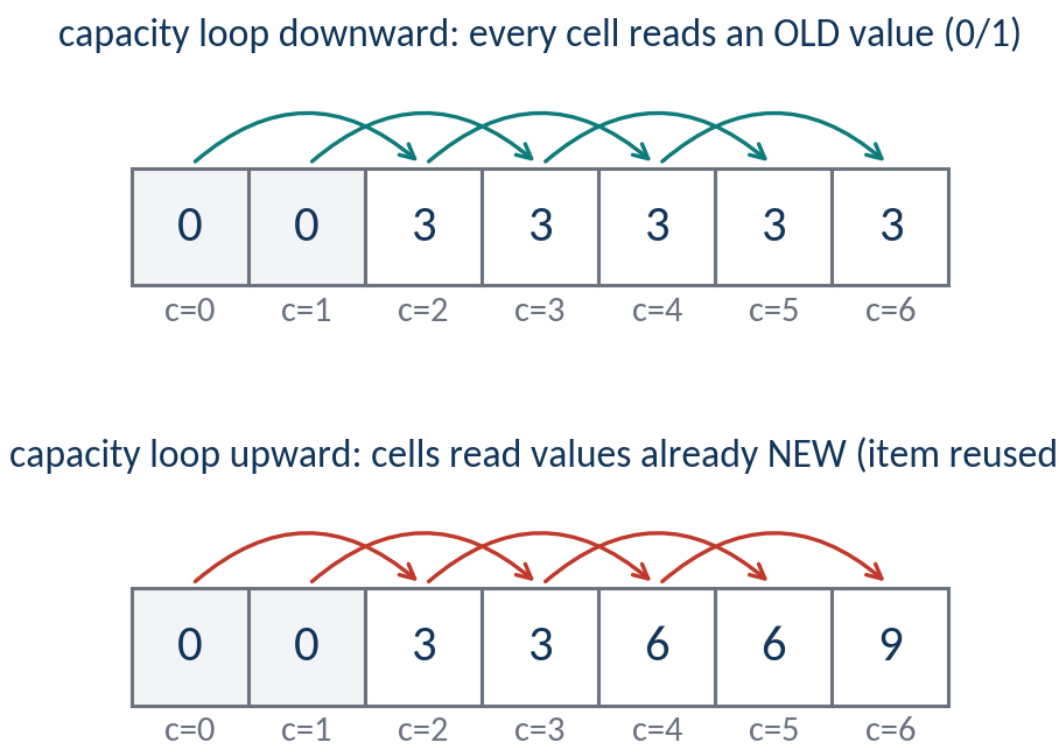


Figure 9.1 One item of size 2 and value 3 added to an all-zero row. Downward, every cell reads a value from before the item, so the item is used at most once; upward, cell 4 reads cell 2 after it already contains the item, so the item is counted twice, and cell 6 three times.

- Looping **downward**, from S to s_i, every cell to the left is still untouched when it is read, so it holds $best_{i-1}$. This is exactly Equation (9.4): each item is taken at most once.
- Looping **upward**, the cell to the left has already been updated for item i, so it may already contain item i, and adding v_i again takes the item a second or third time. That computes a different problem, the *unbounded knapsack*, where every item can be taken any number of times.

Neither direction crashes, and both give the right answer on many small samples, which is why the wrong direction is one of the most common Wrong Answers in DP. **Listing 9.8** runs both directions on four items and checks them against a brute force over all 16 subsets.

Listing 9.8 `knap_direction.cpp`: the two loop directions against a brute force.

```cpp
// What the direction of the capacity loop really computes, checked against
// a brute force over all subsets.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

long long run(const vector<int>& w, const vector<long long>& v, int cap,
              bool down) {
    vector<long long> best(cap + 1, 0);
    for (size_t i = 0; i < w.size(); i++) {
        if (down) {
            for (int c = cap; c >= w[i]; c--)
                best[c] = max(best[c], best[c - w[i]] + v[i]);
        } else {
            for (int c = w[i]; c <= cap; c++)
                best[c] = max(best[c], best[c - w[i]] + v[i]);
        }
    }
    return best[cap];
}

long long brute(const vector<int>& w, const vector<long long>& v, int cap) {
    int n = w.size();
    long long ans = 0;
    for (int mask = 0; mask < (1 << n); mask++) {  // every subset of the items
        long long size = 0, value = 0;
        for (int i = 0; i < n; i++)
            if (mask >> i & 1) size += w[i], value += v[i];
        if (size <= cap) ans = max(ans, value);
    }
    return ans;
}

int main() {
    vector<int> w = {1, 3, 4, 5};
    vector<long long> v = {1, 4, 5, 7};
    int cap = 10;
    cout << "brute force over all 16 subsets: " << brute(w, v, cap) << '\n';
    cout << "capacity loop downward (0/1):    " << run(w, v, cap, true) << '\n';
    cout << "capacity loop upward:            " << run(w, v, cap, false) << '\n';
    return 0;
}
```

```
$ ./knap_direction
```

```
brute force over all 16 subsets: 13
capacity loop downward (0/1):    13
capacity loop upward:            14
```

The brute force and the downward loop agree on 13 (sizes 1, 4 and 5, values 1, 5 and 7). The upward loop reports 14, which is two copies of the item with size 5 and value 7: a correct answer to the unbounded problem and a wrong one here. Keep the rule in one sentence: *0/1 means downward; unlimited copies mean upward.*

9.7.2 The programs

Listing 9.9 is Equation (9.4) in one row, with the capacity loop running downward and `long long` values.

Listing 9.9 `knapsack.cpp`: 0/1 knapsack in one row, capacity loop downward.

```cpp
// KNAPSACK (The Knapsack Problem): 0/1 knapsack with a one-row table.
// The capacity loop runs DOWNWARD so that each item is used at most once.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int cap, n;
    if (!(cin >> cap >> n)) return 0;
    vector<long long> best(cap + 1, 0);  // best[c]: most value with total size <= c
    for (int i = 0; i < n; i++) {
        int size;
        long long value;
        cin >> size >> value;
        for (int c = cap; c >= size; c--)  // downward: best[c - size] is still old
            best[c] = max(best[c], best[c - size] + value);
    }
    cout << best[cap] << '\n';
    return 0;
}
```

In Python, a double loop over four million cells takes a few seconds, too slow for a one-second limit. **Listing 9.10** uses a different idea that is both faster and safer: for each item it builds a *whole new row* from the old one with `zip`. Every new cell `max(keep, took + value)` reads only the old row, so the direction problem cannot arise at all, and the loop runs inside the list comprehension, which is much faster than an explicit `for` over indices.

Listing 9.10 `knapsack.py`: the same DP in Python, one new row per item.

```python
"""KNAPSACK (The Knapsack Problem): 0/1 knapsack, one whole row per item.
The new row is built from the OLD row, so each item is used at most once."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    cap, n = int(data[0]), int(data[1])
    best = [0] * (cap + 1)  # best[c]: most value with total size <= c
    for i in range(n):
        size, value = int(data[2 + 2 * i]), int(data[3 + 2 * i])
        if size > cap:
            continue
        # capacities below size keep their value; the rest may add this item
        best = best[:size] + [
            max(keep, took + value) for keep, took in zip(best[size:], best)
        ]
    print(best[cap])


main()
```

The expression `zip(best[size:], best)` pairs `best[c]` with `best[c - size]` for every capacity c from `size` to S; capacities below `size` cannot hold the item and keep their old values, which is the slice `best[:size]`. Items larger than the whole bag are skipped. We run both programs on the four items used above.

```
$ printf '10 4\n1 1\n3 4\n4 5\n5 7\n' | ./knapsack
$ printf '10 4\n1 1\n3 4\n4 5\n5 7\n' | python3 ../python/knapsack.py
```

```
13
13
```

9.8 Pattern 5: Two Strings (EDIST)

The problem in our words. EDIST (Edit distance) [7] gives up to 10 pairs of strings of capital letters, each at most 2000 characters long. For each pair, print the least number of single-letter operations that turn the first string into the second, where an operation inserts a letter, deletes a letter, or replaces one letter by another. This number is the *edit distance*, or *Levenshtein distance* [8]; spelling checkers use it to suggest the words closest to a mistyped one, and biologists use its relatives to compare DNA.

The state is a pair of prefixes. Let $d(i, j)$ be the least number of operations that turn the first i letters of A into the first j letters of B. Look at the last letters, A_i and B_j. Either the last operation deletes A_i, or it inserts B_j, or A_i is matched with B_j, for free when they are equal and at the cost of one replacement when they differ [9]:

$$d(i, j) = \min(d(i-1, j) + 1,\ d(i, j-1) + 1,\ d(i-1, j-1) + [A_i \neq B_j]) \tag{9.5}$$

The base cases are $d(i, 0) = i$ (delete all i letters) and $d(0, j) = j$ (insert all j letters). **Figure 9.2** shows the full table for SUNDAY and SATURDAY, with one optimal path traced back from the corner. The two horizontal steps in the first row insert A and T, the diagonal steps match or replace, and the answer in the corner is 3: insert A, insert T, replace N by R.

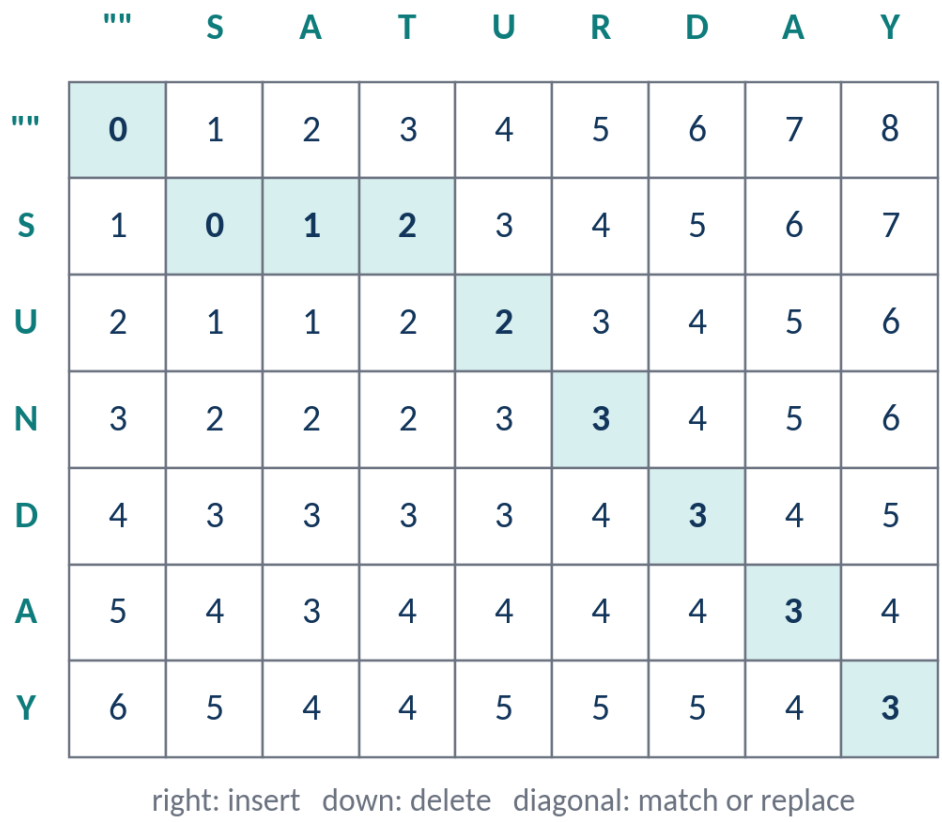


Figure 9.2 The edit-distance table for SUNDAY (rows) and SATURDAY (columns); every cell follows from its left, upper and upper-left neighbors by Equation (9.5), and the shaded cells form one cheapest path with distance 3.

Memory. A full table has 2001×2001 cells: four million `int`s, 16 MB. It fits, but, as in the knapsack, row i needs only row $i-1$ and the cell to its left, so two rows of $m+1$ numbers are enough, and that is what **Listing 9.11** keeps. Keep the full table only when you must print the operations themselves, because tracing a path back needs every cell.

Listing 9.11 `edist.cpp`: edit distance with two rows.

```cpp
// EDIST (Edit distance): fewest insertions, deletions and replacements that
// turn string A into string B. Two-string DP kept in two rows.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;
    while (t--) {
        string a, b;
        cin >> a >> b;
        int n = a.size(), m = b.size();
        vector<int> prev(m + 1), cur(m + 1);       // prev = row i-1, cur = row i
        for (int j = 0; j <= m; j++) prev[j] = j;  // "" -> b[0..j): j insertions
        for (int i = 1; i <= n; i++) {
            cur[0] = i;  // a[0..i) -> "": i deletions
            for (int j = 1; j <= m; j++) {
                int replace = prev[j - 1] + (a[i - 1] != b[j - 1]);
                cur[j] = min({replace, prev[j] + 1, cur[j - 1] + 1});
            }
            swap(prev, cur);
        }
        cout << prev[m] << '\n';
    }
    return 0;
}
```

Listing 9.12 `edist.py`: the same DP in Python, with a lean inner loop.

```python
"""EDIST (Edit distance): fewest insertions, deletions and replacements
that turn A into B. Two rows, with the inner loop kept as lean as possible."""

import sys


def distance(a: str, b: str) -> int:
    prev = list(range(len(b) + 1))  # "" -> b[:j] needs j insertions
    for i, ch in enumerate(a, 1):
        cur = [i]  # a[:i] -> "" needs i deletions
        left = i
        for diag, up, bc in zip(prev, prev[1:], b):
            x = diag if ch == bc else diag + 1  # match or replace
            if up + 1 < x:
                x = up + 1  # delete ch
            if left + 1 < x:
                x = left + 1  # insert bc
            cur.append(x)
            left = x
        prev = cur
    return prev[-1]


def main() -> None:
    data = sys.stdin.read().split()
    t = int(data[0])
    out = [str(distance(data[1 + 2 * k], data[2 + 2 * k])) for k in range(t)]
    print("\n".join(out))


main()
```

The Python inner loop looks different from the C++ one on purpose; Section 9.11 measures why. It walks over `zip(prev, prev[1:], b)`, which hands it the upper-left value, the upper value and the letter of B together without any indexing, keeps the left neighbor in a local variable, and replaces the three-argument `min` by two comparisons.

```
$ printf '2\nFOOD\nMONEY\nSUNDAY\nSATURDAY\n' | ./edist
$ printf '2\nFOOD\nMONEY\nSUNDAY\nSATURDAY\n' | python3 ../python/edist.py
```

```
4
3
4
3
```

FOOD becomes MONEY in 4 operations (for example replace F by M, replace O by N, replace D by E, insert Y), and SUNDAY becomes SATURDAY in the 3 operations of **Figure 9.2**.

9.9 Pattern 6: Intervals (MIXTURES)

The problem in our words. In MIXTURES [10], $n \le 100$ mixtures stand in a row, each with a color from 0 to 99. Two neighboring mixtures can be mixed into one; mixing colors a and b gives the color $(a + b)$ mod 100 and produces $a \cdot b$ units of smoke. Mix everything into one mixture with as little smoke as possible. Test cases follow one another until the end of the input.

The order of mixing matters. With colors 30, 50 and 70, mixing 30 and 50 first gives color 80 and 1500 smoke, and then $80 \times 70 = 5600$ more: 7100 in total. Mixing 50 and 70 first gives color 20 and 3500 smoke, then $30 \times 20 = 600$ more: 4100 in total.

The state is an interval. Whatever the order, the mixtures i to j eventually become one mixture, and its color is always $(a_i + \cdots + a_j)$ mod 100, whatever the order; call it $col(i, j)$. The *last* mix that produced it joined a left part $i..k$ and a right part $k + 1..j$ for some split point k, and each part was mixed on its own before that. So

$$cost(i,j) = \min_{i \le k < j} (cost(i,k) + cost(k+1,j) + col(i,k) \cdot col(k+1,j)), \qquad cost(i,i) = 0 \tag{9.6}$$

and the answer is $cost(1, n)$. This is the same shape as the matrix-chain multiplication problem in the textbooks [2]: an interval, a split point, and a cost for joining the two halves.

The order is the new lesson of this pattern. $cost(i, j)$ needs values of *shorter* intervals only, so the table is filled by increasing length: first all intervals of length 2, then 3, and so on. Filling by increasing i would read $cost(k + 1, j)$ before it has been computed. With prefix sums, each $col(i, k)$ costs $O(1)$; there are $O(n^2)$ intervals and $O(n)$ split points each, so $O(n^3)$ work, about 1.7×10^5 steps for $n = 100$. **Listing 9.13** fills the table by length; **Listing 9.14** does the same in Python and precomputes the colors of all intervals.

Listing 9.13 `mixtures.cpp`: interval DP filled by increasing length.

```cpp
// MIXTURES: mix a row of colored mixtures, two neighbors at a time;
// mixing a and b gives color (a+b) mod 100 and a*b smoke. Least total smoke.
// Interval DP: cost[i][j] is the least smoke to merge mixtures i..j into one.
#include <algorithm>
#include <climits>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    while (cin >> n) {              // test cases until end of file
        vector<int> pre(n + 1, 0);  // prefix sums of the colors
        for (int i = 0; i < n; i++) {
            int x;
            cin >> x;
            pre[i + 1] = pre[i] + x;
        }
        auto color = [&](int i, int j) { return (pre[j + 1] - pre[i]) % 100; };
        vector<vector<long long>> cost(n, vector<long long>(n, 0));
        for (int len = 2; len <= n; len++) {  // shorter intervals first
            for (int i = 0; i + len - 1 < n; i++) {
                int j = i + len - 1;
                long long best = LLONG_MAX;
                for (int k = i; k < j; k++)  // the last mix joins i..k with k+1..j
                    best = min(best, cost[i][k] + cost[k + 1][j] +
                                     color(i, k) * color(k + 1, j));
                cost[i][j] = best;
            }
        }
        cout << cost[0][n - 1] << '\n';
    }
    return 0;
}
```

Listing 9.14 `mixtures.py`: the same interval DP in Python, with the colors of all intervals precomputed.

```python
"""MIXTURES: mix neighbors two at a time; a and b give color (a+b) mod 100
and a*b smoke. Interval DP over cost[i][j], test cases until end of file."""

import sys


def least_smoke(colors: list[int]) -> int:
    n = len(colors)
    pre = [0]
    for x in colors:
        pre.append(pre[-1] + x)
    col = [[(pre[j + 1] - pre[i]) % 100 for j in range(n)] for i in range(n)]
    cost = [[0] * n for _ in range(n)]
    for length in range(2, n + 1):  # shorter intervals first
        for i in range(n - length + 1):
            j = i + length - 1
            ci, cj = cost[i], col[i]
            cost[i][j] = min(
                ci[k] + cost[k + 1][j] + cj[k] * col[k + 1][j] for k in range(i, j)
            )
    return cost[0][n - 1]


def main() -> None:
    data = sys.stdin.buffer.read().split()
    pos, out = 0, []
    while pos < len(data):
        n = int(data[pos])
        colors = list(map(int, data[pos + 1 : pos + 1 + n]))
        pos += 1 + n
        out.append(str(least_smoke(colors)))
    print("\n".join(out))


main()
```

```
  $ printf '3\n30 50 70\n4\n10 20 30 40\n' | ./mixtures
  $ printf '3\n30 50 70\n4\n10 20 30 40\n' | python3 ../python/mixtures.py
```

```
4100
3500
4100
3500
```

The first test gives the 4100 found above. Reading “until the end of the input” is done by `while (cin >> n)` in C++ and by walking through the list of words in Python; Chapter 2 showed why a loop that waits for a sentinel line would hang here.

9.10 Testing It: Every Pattern Against a Brute Force

Chapter 8 introduced stress testing: generate many small random inputs, compute the answer with a slow program that is obviously right, and compare. For DP this is especially valuable, because DP bugs (a wrong base case, a wrong loop direction, an off-by-one in an index) usually show up only on some inputs. **Table 9.1** lists the brute force used for each problem. None of them shares any idea with the DP it checks: they enumerate every choice directly.

Table 9.1 The brute forces and small inputs used to check the six DP programs.

Problem	Brute force in `tests/brute.py`	Small inputs from `tests/cases.py`
FARIDA	every subset of monsters without two neighbors	up to 5 cases, $0 \le N \le 10$
ACODE	recursion over the first letter, no memo	up to 5 strings of 1 to 12 digits
BYTESM2	every walk from every top tile	grids up to 5×5
KNAPSACK	every subset of items	up to 10 items, some larger than the bag
EDIST	plain recursion on the first letters	strings of 1 to 6 letters over A, B, C
MIXTURES	every order of mixing neighbors	up to 6 mixtures

The driver, **Listing 9.15**, compiles each C++ program, runs it and the Python program on the same random input, and compares both outputs with the brute force. On the first difference it prints the failing input, which is small enough to trace by hand.

Listing 9.15 `tests/stress.py`: the stress-test driver for this chapter's six problems.

```
"""Stress test: random small inputs; every C++ and Python solution is
compared with the brute force. Usage: python stress.py [rounds] [seed]"""

import os
import random
import subprocess
import sys

import brute
import cases

HERE = os.path.dirname(os.path.abspath(__file__))
ROOT = os.path.dirname(HERE)
EXE = ".exe" if os.name == "nt" else ""
PROBLEMS = ["farida", "acode", "bytesm2", "knapsack", "edist", "mixtures"]


def build(name: str) -> str:
    os.makedirs(os.path.join(HERE, "build"), exist_ok=True)
    exe = os.path.join(HERE, "build", name + EXE)
    src = os.path.join(ROOT, "cpp", name + ".cpp")
    subprocess.run(["g++", "-O2", "-std=c++17", "-o", exe, src], check=True)
    return exe


def run(cmd: list[str], text: str) -> str:
    done = subprocess.run(cmd, input=text, capture_output=True, text=True)
    return done.stdout.strip()


def main() -> None:
    rounds = int(sys.argv[1]) if len(sys.argv) > 1 else 200
    rng = random.Random(int(sys.argv[2]) if len(sys.argv) > 2 else 2026)
    failures = 0
    for name in PROBLEMS:
        exe = build(name)
        py = [sys.executable, os.path.join(ROOT, "python", name + ".py")]
        bad = 0
        for _ in range(rounds):
            text = getattr(cases, name)(rng)
            want = getattr(brute, name)(text)
            for who, cmd in (("C++", [exe]), ("Python", py)):
                got = run(cmd, text)
                if got != want and bad == 0:  # show the first failure only
                    print(f"{name}: {who} differs on\n{text}", end="")
                    print(f"want {want!r}, got {got!r}")
```

Listing 9.15 (Continued) *tests/stress.py: the stress-test driver for this chapter's six problems.*

```
                bad += got != want
        failures += bad
        print(
            f"{name:9s} {rounds} random inputs: "
            + (
                "C++ and Python agree with brute force"
                if bad == 0
                else f"{bad} wrong answers"
            )
        )
    sys.exit(1 if failures else 0)


if __name__ == "__main__":
    main()
```

```
$ cd tests
$ python3 stress.py 300
```

```
farida    300 random inputs: C++ and Python agree with brute force
acode     300 random inputs: C++ and Python agree with brute force
bytesm2   300 random inputs: C++ and Python agree with brute force
knapsack  300 random inputs: C++ and Python agree with brute force
edist     300 random inputs: C++ and Python agree with brute force
mixtures  300 random inputs: C++ and Python agree with brute force
```

All 1800 random inputs, each with several test cases, give identical answers in the three programs. The small inputs deliberately include the edge cases discussed above: rows with no monsters, items larger than the bag, strings with zeros, grids of one row or one column.

9.11 Making It Better: Speed in Both Languages

Correct is not yet Accepted. `tests/timing.py` builds the largest input each problem allows (for EDIST, ten pairs of 2000-letter strings) and measures the CPU time of every program. **Table 9.2** summarizes the result on our machine, a Linux PC with g++ 13 and Python 3.11.

```
program          C++ (s)  Python (s)  same output
farida              0.01        0.01  yes
acode               0.01        0.03  yes
bytesm2             0.01        0.01  yes
knapsack            0.00        0.19  yes
edist               0.10        3.46  yes
mixtures            0.01        0.19  yes
edist_plain         0.10        7.78  yes
```

Table 9.2 Largest inputs and measured CPU time; the last column is our judgment of how much room Python has.

Problem	Work on the largest input	C++ (s)	Python (s)	Python on SPOJ
FARIDA	10^4 steps	0.01	0.01	comfortable
ACODE	5000 steps	0.01	0.03	comfortable
BYTESM2	10^4 steps	0.01	0.01	comfortable
KNAPSACK	4×10^6 cells	0.00	0.19	only with whole-row updates
EDIST	4×10^7 cells	0.10	3.46	risky: judges are often slower
MIXTURES	1.7×10^6 steps	0.01	0.19	comfortable

Two lessons stand out. First, C++ has room to spare everywhere; the DP ideas, not the language, are what make these problems solvable. Second, Python needs care exactly where the table is

large. For EDIST the first Python version, `python/edist_plain.py`, writes Equation (9.5) directly with indices and `min`; it needs 7.78 seconds on the largest input. The lean loop of **Listing 9.12** computes the same numbers in 3.46 seconds, less than half, by removing every list index and every call to `min` from the inner loop. The EDIST limit is generous, but judges often run on slower processors than a modern PC; for a table of 40 million cells, choose C++.

Three general ways to make a DP faster or smaller follow from this chapter:

- **Roll the table.** When row i needs only row $i-1$, keep two rows (or one, with the right loop direction). FARIDA needs two numbers, BYTESM2 and EDIST two rows, KNAPSACK one row.
- **Precompute what the inner loop repeats.** MIXTURES uses prefix sums for the colors; the Python version even precomputes every $col(i, j)$.
- **Move the loop out of the interpreter.** In Python, whole-row list comprehensions with `zip` run several times faster than index loops, as KNAPSACK shows.

9.12 Using It for Real

The six patterns cover a surprising share of the DP problems you will meet on SPOJ and elsewhere. **Table 9.3** puts them side by side, so that a new problem can be matched to a known shape.

Table 9.3 Six DP patterns: the state, the last decision that gives the recurrence, and the order of filling.

Pattern	State	Last decision	Order	Example
Line of choices	prefix length i	use item i or not	i increasing	FARIDA
Counting ways	prefix length i	length of the last piece	i increasing	ACODE
Grid paths	cell (r, c)	where the last step came from	row by row	BYTESM2
Items under a budget	items seen i, budget c	take item i or not	i increasing, c downward	KNAPSACK
Two sequences	prefixes i and j	delete, insert or match	row by row	EDIST
Intervals	interval $[i, j]$	the last split point k	length increasing	MIXTURES

Changing the rules changes the state, and the five questions tell you how. Suppose Bu Wati starts selling durian at some markets, and after a durian market she needs *two* days at home, because durian must be collected at night when it falls. The last decision is still "sell on day i or not", but selling durian on day i now forbids days $i-1$ and $i-2$, so that term of the recurrence reads $best(i-3)+c_i$ instead of $best(i-2)+c_i$, while the other days keep the old rule. The program changes in one line and keeps three rolling values instead of two. If instead she may sell on at most K days in the two weeks, one number no longer describes a subproblem: the state becomes $best(i, j)$, the most she can earn in the first i days using exactly j market days, a table of $n \times K$ values. Asking "what does a subproblem need to remember?" is how the state grows exactly as much as the rules require, and no more.

When a problem of your own arrives, whether on a judge or at work, go through the five questions of Section 9.2.3 on paper before writing code:

1. Write one sentence that defines the state, with units: "$best(i, c)$ is the most value from the first i items with total size at most c." If you cannot write this sentence, you do not have a DP yet.
2. Ask "what was the last decision?" and write one term of the recurrence for each possible answer.

3. Write the base cases and check them on the smallest input: $n = 0$, an empty string, a single tile.
4. Draw two or three cells of the table with arrows to the cells they read, and choose the order so that every arrow points to a cell already filled.
5. Multiply the number of states by the work per state and compare it with the limits.

The same method applies far from contests. A small shop deciding which orders to accept with a limited number of working hours is a knapsack. A clinic comparing two spellings of a patient's name in two registers is computing an edit distance. A farmer planning which beds to plant when neighboring beds must rest is Bu Wati's problem again. In each case the program is short once the state is named. For many more patterns (DP over subsets with bitmasks, over digits, over trees), the catalogs of Halim, Halim and Effendy [11] and of Laaksonen [12] are the natural next step; both are organized, like this chapter, by the shape of the state.

9.13 Common Mistakes, and How to Avoid Them

- **A vague state.** "dp[i] is the answer up to i" hides the question: using item i or not? Including day i or ending at day i? Write the state as one precise sentence before writing the recurrence; most wrong recurrences come from a state that means two things.
- **Missing or wrong base cases.** Forgetting $ways(0) = 1$ makes every ACODE count 0; setting $best(1)$ from an empty array crashes FARIDA when $N = 0$. Test the smallest inputs first.
- **Filling in the wrong order.** In interval DP, loop over the length first; in any DP, check that every value is written before it is read.
- **The wrong loop direction in a one-row knapsack.** Upward computes the unbounded knapsack. For 0/1, loop downward, or build a new row from the old one.
- **Overflow.** Totals of many large values (FARIDA up to 10^{13}) and counts of ways (ACODE) outgrow `int`. Estimate the largest possible value before choosing the type.
- **A memo sentinel that can be a real answer.** Using 0 for "not computed" when 0 is a possible answer makes the memo useless and the program exponential again. Use -1 when answers are non-negative, or a separate flag table.
- **Deep recursion.** A top-down DP over 10^5 states can overflow the C++ stack and exceeds Python's default limit. Prefer bottom-up for long chains.
- **State left over from the previous test case.** A table declared once and reused must be reset for every test case (Chapter 7). Declaring the rows inside the test-case loop, as all programs here do, removes the danger.

9.14 Chapter Summary

- Dynamic programming solves a problem by naming its subproblems (the *state*), expressing each state through smaller ones (the *recurrence*), answering the smallest directly (the *base cases*), and filling a table in an *order* in which every value is ready when it is read.
- Plain recursion on a recurrence repeats the same subproblems exponentially often; a memo (top-down) or a table (bottom-up) computes each state once. **Listing 9.1** showed 331 million calls shrink to 79.
- Six patterns cover much of SPOJ: a line of choices (FARIDA, Equation (9.1)), counting ways (ACODE, Equation (9.2)), grid paths (BYTESM2, Equation (9.3)), items under a budget (KNAPSACK, Equation (9.4)), two strings (EDIST, Equation (9.5)) and intervals (MIXTURES, Equation (9.6)).

- The running time is the number of states times the work per state; memory can often be cut to one or two rows. In a one-row 0/1 knapsack the capacity loop must run downward.
- Every program was checked against brute forces on 1800 random inputs and timed on the largest inputs. C++ has ample room everywhere; Python is comfortable except for the largest tables, where whole-row operations and lean inner loops help, and C++ remains the safe choice.

9.15 Exercises

1. By hand, apply Equation (9.1) to the coins 2,7,9,3,1. Give every value of $best(i)$, the answer and the monsters taken. Check the answer with `farida.cpp`.
2. By hand, count the readings of the ACODE string 2101 with Equation (9.2). Why does the digit 0 make the count *smaller* than for 2111?
3. In how many ways can you pay 10 with coins of 1, 2 and 5, if the order of the coins does not matter and every coin may be used any number of times? Write the one-row recurrence, say in which direction its loop must run, and fill the row by hand.
4. For MIXTURES with the colors 10, 20, 30, 40, the program prints 3500. Find an order of mixing that produces exactly 3500 smoke, and explain why the table must be filled by interval length.
5. By hand, fill the edit-distance table for KITTEN and SITTING and give a list of operations that achieves the distance.
6. Bu Wati wants to know not only how much she can earn but also *which* market days to choose. Describe how to recover the chosen days from the table of Equation (9.1), as pseudocode, and apply it to the days of Section 9.2.2.

9.16 Solutions to the Exercises

1. $best(0) = 0$, $best(1) = 2$, $best(2) = \max(2{,}0 + 7) = 7$, $best(3) = \max(7{,}2 + 9) = 11$, $best(4) = \max(11{,}7 + 3) = 11$, $best(5) = \max(11{,}11 + 1) = 12$. The answer is 12, from monsters 1, 3 and 5 (coins 2, 9 and 1). The program agrees:

```
$ printf '1\n5\n2 7 9 3 1\n' | ./farida
Case 1: 12
```

2. $ways(0) = 1$, $ways(1) = 1$ (B), $ways(2) = 1 + 1 = 2$ (B A, U), $ways(3) = 0 + ways(1) = 1$, because the digit 0 cannot stand alone and only “10” is possible, and $ways(4) = ways(3) + 0 = 1$, because “01” is not a letter. The single reading is B J A. For 2111 every digit can stand alone and every pair (21, 11, 11) is a letter, so the count is 1,1,2,3,5: five readings. A 0 removes options on both sides of it, since it must be glued to the digit before it.

3. Let $ways(a)$ be the number of ways to pay a with the coins considered so far, $ways(0) = 1$. Processing one coin value v at a time, $ways(a) += ways(a - v)$ for a from v up to 10. The loop runs *upward*, because a coin may be used again: exactly the “unbounded” direction of Section 9.7.1. Processing the coins one value at a time, in an outer loop, is what makes the order of the coins irrelevant (each combination is counted once, in the order 1s, then 2s, then 5s). After the 1s, every $ways(a) = 1$. After the 2s, $ways(a) = \lfloor a/2 \rfloor + 1$, so $ways(0), \ldots, ways(10) =$ 1,1,2,2,3,3,4,4,5,5,6. After the 5s, going upward, first $ways(5) = 3 + ways(0) = 4$, and later $ways(10) = 6 + ways(5) = 6 + 4 = 10$, where $ways(5)$ already counts the ways that use a 5. There are 10 ways.

4. Mix 10 and 20 (smoke 200, color 30), then this 30 with the next 30 (smoke 900, color 60), then 60 with 40 (smoke 2400, color 0): $200 + 900 + 2400 = 3500$. The table must be filled by length

because $cost(i, j)$ in Equation (9.6) reads $cost(k + 1, j)$, an interval that starts to the right of i; with rows filled by increasing i it would still be empty. Every interval it reads is shorter, so filling by length is always safe.

```
$ printf '4\n10 20 30 40\n' | ./mixtures
```

```
3500
```

5. With KITTEN as rows and SITTING as columns, the last row of the table is 6,6,5,4,3,3,2,3 and the corner holds 3. One cheapest list: replace K by S, replace E by I, insert G. The program agrees:

```
$ printf '1\nKITTEN\nSITTING\n' | ./edist
```

```
3
```

6. Keep the whole table $best(0..n)$ and walk back from the end. At day i, if $best(i) = best(i - 1)$, day i was not needed; otherwise day i was chosen and the walk jumps to $i - 2$. **Pseudocode 9.1** writes this down.

Pseudocode 9.1 Recovering the chosen days from the FARIDA table by walking back from the last day.

```
TRACE-DAYS(best, c, n)
1  days = empty list
2  i = n
3  while i >= 1
4      if i >= 2 and best[i] == best[i-1]
5          i = i - 1                  // day i is not needed
6      elseif i == 1 and best[1] == 0
7          i = 0                      // nothing earned on day 1
8      else
9          prepend i to days          // day i was chosen
10         i = i - 2
11 return days
```

For $c = (4,1,1,9,2)$ and $best = (0,4,4,5,13,13)$: at $i = 5$, $best(5) = best(4)$, so go to 4; at $i = 4$, $13 \neq 5$, so day 4 is chosen and we jump to 2; at $i = 2$, $best(2) = best(1)$, so go to 1; at $i = 1$, $best(1) = 4 > 0$, so day 1 is chosen. The days are 1 and 4, as in Section 9.2.2. The same "walk back through the table" recovers the chosen items of a knapsack (keep the full two-dimensional table) and the operations of an edit distance (follow the path of **Figure 9.2**).

References

[1] Sphere Online Judge, "FARIDA - Princess Farida." [Online]. Available: https://www.spoj.com/problems/FARIDA/

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 14.

[3] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 1957.

[4] Sphere Online Judge, "ACODE - Alphacode." [Online]. Available: https://www.spoj.com/problems/ACODE/

[5] Sphere Online Judge, "BYTESM2 - Philosophers Stone." [Online]. Available: https://www.spoj.com/problems/BYTESM2/

[6] Sphere Online Judge, "KNAPSACK - The Knapsack Problem." [Online]. Available: https://www.spoj.com/problems/KNAPSACK/

[7] Sphere Online Judge, "EDIST - Edit distance." [Online]. Available: https://www.spoj.com/problems/EDIST/

[8] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[9] 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.

[10] Sphere Online Judge, "MIXTURES - Mixtures." [Online]. Available: https://www.spoj.com/problems/MIXTURES/

[11] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Lulu Press, 2020.

[12] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

Chapter 10

Greedy Algorithms and Exchange Arguments

A *greedy* algorithm builds its answer one piece at a time and, at every step, takes the piece that looks best right now, never going back. When it works, it gives the shortest and fastest programs in this book: usually a sort and one loop. The danger is that it often *looks* right when it is wrong. This chapter teaches you to tell the two apart. By the end you will be able to state a greedy rule precisely, prove it with an *exchange argument* (take any optimal answer and swap it, step by step, into the greedy one without making it worse), find counterexamples to wrong rules with a brute-force checker, and write clean C++17 and Python 3 solutions for four real SPOJ problems: BUSYMAN, FASHION, BAISED and GERGOVIA. Along the way you will see why a cashier's "largest coin first" is right for rupiah but wrong for other coin systems, and why the knapsack of Chapter 9 needs DP while its fractional cousin does not.

10.1 Pak Harun's Community Hall

Pak Harun looks after the community hall of his *kampung*. Every month the requests pile up in his notebook: a wedding reception from ten in the morning to four in the afternoon, the women's *arisan* from one to three, a youth-group meeting from four to six, a Quran recitation from seven to nine in the evening, a birthday party from eight to ten. The hall can hold only one event at a time, and the neighborhood committee has asked him for one thing: host *as many* events as possible, so that as many families as possible are served. Money is not the question; the count is.

He tries a few rules of thumb. "First come, first served by starting time" gives the hall to the wedding at ten and blocks everything until four. "Shortest events first" sounds fair, but a short event in the middle of the afternoon can block two longer ones that would have fitted before and after it. Each rule is simple, each sounds reasonable, and some of them are wrong. Which one is right, and how can Pak Harun be *sure*?

The same question, with activities instead of events, is the SPOJ problem BUSYMAN (I AM VERY BUSY) [1]. It is the classic first example of a greedy algorithm [2], and it is also the classic example of how to prove one. Once you have seen the proof, you will recognize the same argument in pairing problems, ranking problems and transport problems, and you will know what to look for when a greedy rule is *not* correct.

10.2 The Idea in Plain Words

A greedy algorithm makes a sequence of choices. At each step it applies a fixed rule, such as "take the event that ends earliest among those that still fit", and it never undoes a choice. That makes it fast, but it also makes it fragile: one bad early choice can never be repaired.

> **DEFINITION.** A *greedy algorithm* builds a solution step by step, each time adding the element that is best according to a fixed local rule, and never revises an earlier choice. A greedy rule is *correct* for a problem if the solution it builds is optimal for every valid input. An *exchange*

> *argument* proves correctness by taking an arbitrary optimal solution and transforming it, one exchange at a time, into the greedy solution, showing that no exchange makes it worse [3].

There are two standard ways to prove that a greedy rule is correct [2], [3]:

- **Exchange argument.** Take any optimal solution O. If it differs from the greedy solution G, find the first place where they differ and change O there to agree with G. Show that the changed solution is still valid and still optimal. Repeating the step turns O into G, so G is optimal too.
- **Greedy stays ahead.** Measure progress after each step, for example "the time at which the hall becomes free after the k-th event". Show by induction that after every step the greedy solution is at least as far ahead as any other solution. Then the greedy solution cannot finish behind.

And there is one standard way to show that a greedy rule is *wrong*: a single counterexample. Chapter 8 taught you to find counterexamples automatically with a brute force and random tests; greedy rules are the best customers of that technique.

> **KEY INSIGHT:** A greedy algorithm is easy to write and easy to believe. Before you submit one, either prove it with an exchange argument or test it against a brute force on thousands of small inputs. If you can do neither, suspect DP.

10.3 A First Attempt: Change at the Cashier

Every cashier in Indonesia runs a greedy algorithm many times a day. To give 7,700 rupiah in change, she takes the largest note or coin that is not more than what is still owed, again and again: 5,000, then 2,000, then 500, then 200. The money in everyday use comes in notes of 1,000, 2,000, 5,000, 10,000, 20,000, 50,000 and 100,000 rupiah [4] and coins of 100, 200, 500 and 1,000 rupiah [5]. Does "largest first" always give the *fewest* pieces?

For the rupiah the answer is yes, but that is a property of the denominations, not of greedy algorithms. Consider a made-up coin system with coins of 1, 3 and 4. To pay 6, greedy takes 4, then 1, then 1: three coins. But 3 + 3 uses only two. The first choice, the 4, looked best and was wrong, and greedy never takes it back.

Listing 10.1 checks both systems against the true minimum, which it computes with the DP of Chapter 9: $best(a) = \min_{c \le a}(best(a-c)+1)$ over all coins c, with $best(0) = 0$. For the rupiah it tests every multiple of 100 up to 300,000.

Listing 10.1 `coin_change.cpp`: greedy change-making against the minimum computed by DP, for two coin systems.

```cpp
// Greedy change-making ("largest coin first") against the true minimum
// found by DP, for two coin systems: a made-up one and rupiah money.
#include <algorithm>
#include <climits>
#include <iostream>
#include <vector>
using namespace std;

int greedyCount(const vector<int>& coins, int amount) {  // coins sorted descending
    int used = 0;
    for (int c : coins) {
        used += amount / c;  // take as many of the largest coin as fit
        amount %= c;
    }
    return amount == 0 ? used : INT_MAX;
}

vector<int> fewestCoins(const vector<int>& coins, int limit) {  // DP, all amounts
    vector<int> best(limit + 1, INT_MAX);
    best[0] = 0;
    for (int a = 1; a <= limit; a++)
        for (int c : coins)
            if (c <= a && best[a - c] != INT_MAX)
                best[a] = min(best[a], best[a - c] + 1);
    return best;
}

void compare(const char* name, vector<int> coins, int unit, int limit) {
    sort(coins.rbegin(), coins.rend());
    vector<int> best = fewestCoins(coins, limit);
    int bad = 0, first = -1;
    for (int a = unit; a <= limit; a += unit) {
        if (greedyCount(coins, a) != best[a]) {
            bad++;
            if (first < 0) first = a;
        }
    }
    cout << name << ": amounts checked up to " << limit
         << ", greedy not optimal for " << bad;
    if (first >= 0)
        cout << " (first: " << first << " greedy " << greedyCount(coins, first)
             << " coins, best " << best[first] << ")";
    cout << '\n';
}
```

Listing 10.1 (Continued) *coin_change.cpp: greedy change-making against the minimum computed by DP, for two coin systems.*

```cpp
int main() {
    compare("coins {1, 3, 4}", {1, 3, 4}, 1, 1000);
    compare("rupiah",
            {100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, 100000}, 100,
            300000);
    return 0;
}
```

```
  $ g++ -std=c++17 -O2 -Wall -Wextra -o coin_change coin_change.cpp
  $ ./coin_change
```

```
coins {1, 3, 4}: amounts checked up to 1000, greedy not optimal for 249 (first: 6 greedy 3
coins, best 2)
rupiah: amounts checked up to 300000, greedy not optimal for 0
```

With coins 1, 3 and 4, greedy is not optimal for 249 of the first 1000 amounts, the first of them being 6. With the rupiah it is optimal for every amount checked. Coin systems for which greedy is

always optimal are called *canonical*. Kozen and Zaks showed that if a system is not canonical, a counterexample exists below the sum of its two largest coins [6], so a finite check like ours can settle the question, and Pearson later gave a fast test [7]. For rupiah, our check up to 300,000 goes beyond $100{,}000 + 50{,}000$, so the greedy cashier is proven right. For a new coin system, an arcade's tokens or a game's currency, run the check first.

Why does the rupiah behave so well? Its denominations follow the pattern 1, 2, 5 in every power of ten: 100, 200, 500, then 1,000, 2,000, 5,000, and so on. Inside one decade, every amount from one to nine units is paid by greedy with at most three pieces (for example $9 = 5 + 2 + 2$ and $4 = 2 + 2$), and no cleverer combination does better, because a 5 is the only way to cover five or more units with one piece. Greedy then handles each decade on its own, from the top down. The made-up system 1, 3, 4 breaks this regularity: the 4 is too close to the 3, so taking it can leave an awkward remainder.

There is one more trap, and every cashier knows it. The proof assumes an unlimited supply of every coin. When the drawer runs out of 100-rupiah coins, greedy can fail even with rupiah: to give 600 with only 200s and 500s left, greedy takes the 500 and then cannot pay the remaining 100, while $200 + 200 + 200$ works. With limited supplies, change-making is a bounded knapsack, and the DP of Chapter 9 is the tool again.

The lesson for contests is the same: **a greedy rule is correct because of the structure of the problem, and that structure must be shown.** The next four sections show it for four SPOJ problems.

10.4 Picking Activities (BUSYMAN)

The problem in our words. BUSYMAN [1] gives up to 10 test cases. Each has $N \le 10^5$ activities, each with a start time and an end time between 0 and 10^6, the start strictly before the end. A person can do one activity at a time, and an activity may start exactly when the previous one ends. Print the largest number of activities the person can do.

10.4.1 Which rule?

Three rules come to mind: earliest start first, shortest first, and earliest end first. Instead of arguing, **Listing 10.2** tests all three against a brute force over all subsets, on 10,000 random sets of up to seven activities. Each activity is treated as a half-open interval $[s, e)$, so that [5,7) and [7,9) do not overlap.

Listing 10.2 `greedy_keys.cpp`: three greedy rules for activity selection against a brute force.

```
// Three plausible greedy rules for "most non-overlapping activities",
// each checked against a brute force on 10000 random small cases.
#include <algorithm>
#include <iostream>
#include <random>
#include <vector>
using namespace std;

struct Act {
    int s, e;  // start and end; [s, e) so an end may equal the next start
};

int pickGreedy(vector<Act> a, int rule) {
    if (rule == 0) sort(a.begin(), a.end(), [](Act x, Act y) { return x.s < y.s; });
    if (rule == 1)
        sort(a.begin(), a.end(),
             [](Act x, Act y) { return x.e - x.s < y.e - y.s; });
    if (rule == 2) sort(a.begin(), a.end(), [](Act x, Act y) { return x.e < y.e; });
    vector<Act> chosen;
    for (Act x : a) {
        bool fits = true;
        for (Act y : chosen)
            if (x.s < y.e && y.s < x.e) fits = false;  // the two overlap
        if (fits) chosen.push_back(x);
    }
    return chosen.size();
}

int brute(const vector<Act>& a) {
    int n = a.size(), best = 0;
    for (int mask = 0; mask < (1 << n); mask++) {
        bool ok = true;
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j < n; j++)
                if ((mask >> i & 1) && (mask >> j & 1) && a[i].s < a[j].e &&
                    a[j].s < a[i].e)
                    ok = false;
        if (ok) best = max(best, __builtin_popcount(mask));
    }
    return best;
}
```

Listing 10.2 (Continued) *greedy_keys.cpp: three greedy rules for activity selection against a brute force.*

```
int main() {
    const char* names[] = {"earliest start", "shortest first", "earliest end"};
    mt19937 rng(2026);
    int wrong[3] = {0, 0, 0};
    for (int round = 0; round < 10000; round++) {
        int n = rng() % 7 + 1;
        vector<Act> a(n);
        for (Act& x : a) {
            x.s = rng() % 10;
            x.e = x.s + 1 + rng() % 5;
        }
        int best = brute(a);
        for (int r = 0; r < 3; r++) wrong[r] += pickGreedy(a, r) != best;
    }
    for (int r = 0; r < 3; r++)
        cout << names[r] << ": wrong on " << wrong[r] << " of 10000 cases\n";
    return 0;
}
```

```
$ ./greedy_keys
```

```
earliest start: wrong on 1167 of 10000 cases
shortest first: wrong on 157 of 10000 cases
earliest end: wrong on 0 of 10000 cases
```

The verdict is clear. “Earliest start” fails on more than one case in ten: the activities [0,10), [1,2) and [2,3) make it take the long one and nothing else, while the answer is 2. “Shortest first” fails less often, which makes it more dangerous, because it passes most samples: with [0,5), [4,6) and [5,10) it takes the short [4,6), which overlaps both others, and ends with 1 instead of 2. “Earliest end” never fails. Now we prove that it never will.

10.4.2 Why earliest end is right

Sort the activities by end time and let g_1 be the one that ends first. Take any optimal schedule $O = (o_1, o_2, \dots, o_k)$, listed in time order. Replace o_1 by g_1. Since g_1 ends no later than any activity,

$$e(g_1) \le e(o_1) \le s(o_2) \tag{10.1}$$

so g_1 does not overlap o_2, and the new schedule $(g_1, o_2, \dots, o_k)$ is valid and has the same size k. So there is an optimal schedule that starts with the greedy choice. After g_1, the remaining problem is the same problem on the activities that start at or after $e(g_1)$, and the same argument applies to it. By induction, the greedy schedule has as many activities as an optimal one. This is exactly the exchange argument of Section 10.2; Equation (10.1) is the one inequality it needs.

The same fact reads as “greedy stays ahead”: after k choices, the greedy schedule frees the person no later than any other schedule with k activities, so it always has at least as much time left for the rest. Written out, the induction is short. Let $g_1, g_2, \dots$ be the greedy activities and $o_1, o_2, \dots, o_k$ those of an optimal schedule, both in time order. For $k = 1$, $e(g_1) \le e(o_1)$ because g_1 ends first of all. If $e(g_j) \le e(o_j)$, then o_{j+1} starts at or after $e(o_j)$, hence at or after $e(g_j)$, so o_{j+1} is still available to the greedy at step $j + 1$; the greedy takes the available activity that ends first, so $e(g_{j+1}) \le e(o_{j+1})$. Therefore the greedy never runs out of activities before the optimal schedule does, and it takes at least k of them.

Figure 10.1 runs the rule on six activities. Sorted by end time, the greedy takes [4,5), then [5,7), which starts exactly when the first ends, then [7,9); the other three all start before 9 and are skipped. The answer is 3.

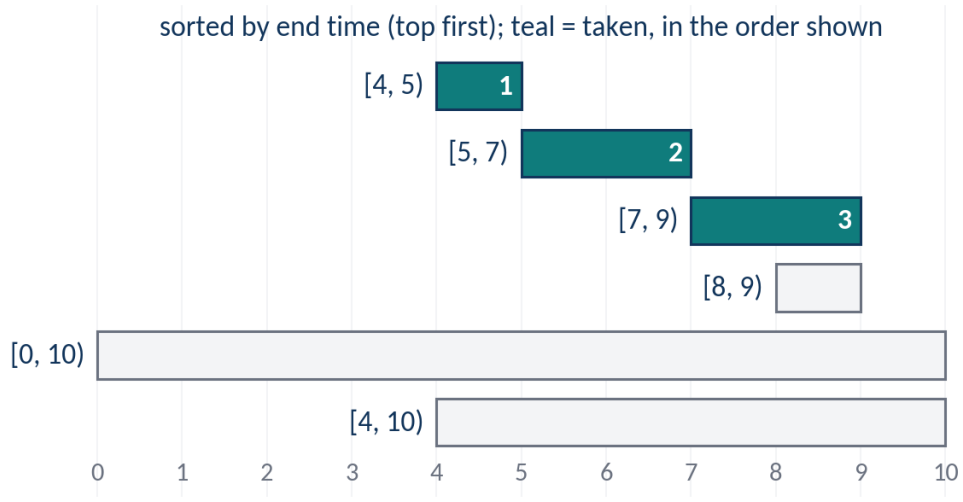


Figure 10.1 Earliest end first on six activities: the three shaded ones are taken in the numbered order; each skipped activity starts before the person is free again.

10.4.3 The programs

In C++, **Listing 10.3** stores each activity as a pair (end, start), so that the standard `sort` orders by end time with no custom comparison; the variable `freeAt` is the time at which the person becomes free. The comparison `s >= freeAt` uses `>=`, not `>`, because an activity may start at the very moment the previous one ends. That one character is the most common Wrong Answer in this problem.

Listing 10.3 `busyman.cpp`: BUSYMAN, sort by end time and take what fits.

```cpp
// BUSYMAN (I AM VERY BUSY): the most activities with no two overlapping.
// Greedy: sort by end time, take every activity that starts after the last end.
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;
    while (t--) {
        int n;
        cin >> n;
        vector<pair<int, int>> act(n);  // stored as (end, start) to sort by end
        for (auto& [e, s] : act) cin >> s >> e;
        sort(act.begin(), act.end());
        int count = 0, freeAt = 0;  // the time at which the person is free again
        for (auto [e, s] : act) {
            if (s >= freeAt) {  // starting exactly when the last one ends is fine
                count++;
                freeAt = e;
            }
        }
        cout << count << '\n';
    }
    return 0;
}
```

Listing 10.4 `busyman.py`: the same greedy in Python.

```python
"""BUSYMAN (I AM VERY BUSY): the most activities with no two overlapping.
Greedy: sort by end time, take each activity that starts after the last end."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    pos, out = 1, []
    for _ in range(t):
        n = int(data[pos])
        starts = map(int, data[pos + 1 : pos + 1 + 2 * n : 2])
        ends = map(int, data[pos + 2 : pos + 2 + 2 * n : 2])
        pos += 1 + 2 * n
        count = free_at = 0  # the time at which the person is free again
        for e, s in sorted(zip(ends, starts)):
            if s >= free_at:  # starting exactly at the last end is fine
                count += 1
                free_at = e
        out.append(str(count))
    print("\n".join(out))


main()
```

Listing 10.4 picks the starts and ends out of the flat list of words with the slices `data[pos + 1:...:2]` and `data[pos + 2:...:2]` and zips them into (end, start) pairs, exactly like the C++ version. We run both on three test cases; the third is the one in **Figure 10.1**.

```
$ printf '3\n3\n3 9\n2 8\n6 9\n4\n1 7\n5 8\n7 8\n1 8\n6\n7 9\n0 10\n4 5\n8 9\n4 10\n5
7\n' | ./busyman
$ printf '3\n3\n3 9\n2 8\n6 9\n4\n1 7\n5 8\n7 8\n1 8\n6\n7 9\n0 10\n4 5\n8 9\n4 10\n5
7\n' | python3 ../python/busyman.py
```

```
1
2
3
1
2
3
```

In the first case all three activities overlap, so only one is possible. In the second, [1,7) ends first and [7,8) starts exactly at 7: two. The third gives the 3 of **Figure 10.1**. Sorting dominates the work: $O(N \log N)$ per test case.

10.4.4 When the events have values

Suppose the committee changes its mind: every event now brings a donation to the hall's cash box, v_j rupiah for event j, and Pak Harun should maximize the total donation instead of the number of events. Earliest end first fails at once: a short meeting worth 50,000 that ends at noon would block a wedding worth 2,000,000 that runs from ten to four. No local rule survives, because the value of taking an event depends on what it blocks.

This *weighted interval scheduling* problem is a DP in the style of Chapter 9 [3]. Sort the events by end time, $e_1 \le e_2 \le \cdots \le e_N$, and let $p(j)$ be the number of events that end no later than event j starts, which binary search on the sorted end times finds in $O(\log N)$ (Chapter 11). The last decision is whether to host event j:

$$best(j) = \max(best(j-1),\ v_j + best(p(j))), \qquad best(0) = 0 \tag{10.2}$$

It is the same shape as the recurrence for Bu Wati's market days in Section 9.2.1, except that "the day before" becomes "the last event that ends in time". When all values are equal, Equation (10.2) and the greedy rule give the same count; the greedy rule is the special case in which the exchange argument works. This pattern repeats throughout the book: a greedy algorithm is often a DP whose choice has been proved in advance.

10.5 Pairing Two Lists (FASHION)

The problem in our words. FASHION (Fashion Shows) [8] gives several test cases. Each has $N \leq 1000$ men and N women, each with a rating from 0 to 10. Pair every man with exactly one woman; a pair with ratings x and y contributes $x \cdot y$. Make the total as large as possible.

The greedy rule is: sort both lists and pair them in order, the lowest man with the lowest woman, and so on up to the highest with the highest. Why?

Suppose some pairing has two *crossed* pairs: men $a_1 \leq a_2$ paired with women b_2 and b_1 respectively, where $b_1 \leq b_2$. Uncross them, pairing a_1 with b_1 and a_2 with b_2. The total changes by

$$(a_1 b_1 + a_2 b_2) - (a_1 b_2 + a_2 b_1) = (a_2 - a_1)(b_2 - b_1) \geq 0 \tag{10.3}$$

because both factors are non-negative. So uncrossing never makes the total smaller. Start from any optimal pairing and uncross crossed pairs one at a time; each exchange reduces the number of crossed pairs, so the process ends, and it ends at the sorted pairing, which therefore is optimal as well. This is the *rearrangement inequality* [9], proved by an exchange argument. **Figure 10.2** shows the smallest case: uncrossing 2 and 7 against 3 and 9 raises the total from 39 to 69, and Equation (10.3) predicts the gain exactly: $(7-2)(9-3) = 30$.

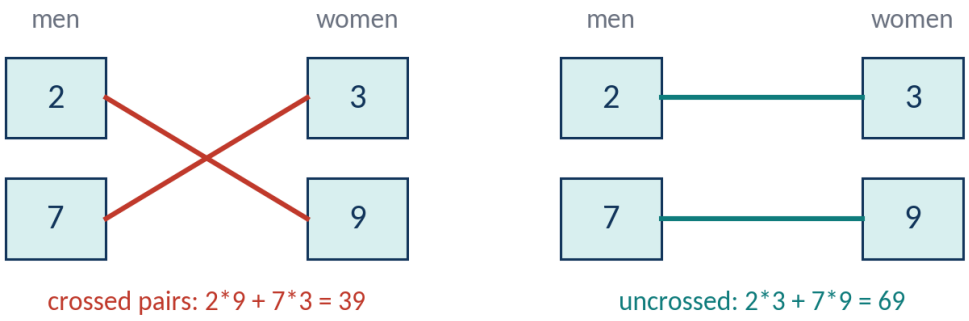


Figure 10.2 Uncrossing two pairs never lowers the sum of products; here it raises it by $(7-2)(9-3) = 30$.

Listing 10.5 `fashion.cpp`: sort both lists and pair them in order.

```cpp
// FASHION (Fashion Shows): pair men and women to maximize the sum of the
// products of their ratings. Sort both lists and pair them in order.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;
    while (t--) {
        int n;
        cin >> n;
        vector<int> men(n), women(n);
        for (int& x : men) cin >> x;
        for (int& x : women) cin >> x;
        sort(men.begin(), men.end());
        sort(women.begin(), women.end());
        long long sum = 0;
        for (int i = 0; i < n; i++) sum += (long long)men[i] * women[i];
        cout << sum << '\n';
    }
    return 0;
}
```

Listing 10.6 `fashion.py`: the same pairing in Python.

```python
"""FASHION (Fashion Shows): pair two lists of ratings to maximize the sum
of products. Sort both lists and pair them in order."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    pos, out = 1, []
    for _ in range(t):
        n = int(data[pos])
        men = sorted(map(int, data[pos + 1 : pos + 1 + n]))
        women = sorted(map(int, data[pos + 1 + n : pos + 1 + 2 * n]))
        pos += 1 + 2 * n
        out.append(str(sum(m * w for m, w in zip(men, women))))
    print("\n".join(out))


main()
```

Listing 10.5 converts one factor to `long long` before multiplying. With ratings up to 10 and $N \le 1000$, the total is at most 10^5 and `int` would suffice, but the cast costs nothing and protects the program when the same code is reused for larger values; **Listing 10.6** needs no such care.

```
$ printf '1\n4\n3 1 4 1\n5 9 2 6\n' | ./fashion
$ printf '1\n4\n3 1 4 1\n5 9 2 6\n' | python3 ../python/fashion.py
```

```
61
61
```

Sorted, the men are 1,1,3,4 and the women 2,5,6,9, and the total is $2 + 5 + 18 + 36 = 61$.

10.6 Places for Wishes (BAISED)

The problem in our words. In BAISED (Biased Standings) [10], N teams (up to about 10^5) each name the place from 1 to N where they would like to finish; several teams may want the same place. The organizers must give every team a different place from 1 to N. The *badness* of a ranking is the sum, over all teams, of the distance between the wished place and the given place. Print the smallest possible badness. The input starts with the number of test cases, each test case is preceded by a blank line, and every team has a name before its wish.

The greedy rule: sort the wishes, $w_{(1)} \le w_{(2)} \le \cdots \le w_{(N)}$, and give place i to the team with the i-th smallest wish. Then

$$badness = \sum_{i=1}^{N} \left| w_{(i)} - i \right| \tag{10.4}$$

Why. The argument is the exchange of Section 10.5 with distances instead of products. Take two teams with wishes $w_1 \le w_2$ that receive places $p_1 > p_2$, a crossed assignment. Swapping their places never increases $|w_1 - p| + |w_2 - q|$: when two points on a line are matched with two other points, the uncrossed matching is never longer than the crossed one. (Check the cases on a number line: if both wishes lie below both places, both matchings cost the same; otherwise the crossed matching travels over some stretch of the line twice.) Uncrossing every crossed pair leads from any optimal ranking to the sorted one, so Equation (10.4) is optimal.

Pitfalls. The badness can reach about $N^2/2 = 5 \times 10^9$ when all teams want place 1, which overflows `int`, so it is summed in `long long` [10]. The blank lines and the team names look scary but are harmless: `cin >>` skips all whitespace, including empty lines, and the names are read into a throwaway string. In Python the names are skipped by taking every second word. **Listing 10.7** sorts the wishes with `sort` and adds up Equation (10.4).

Listing 10.7 `baised.cpp`: sort the wishes and sum the distances.

```cpp
// BAISED (Biased Standings): give every team a distinct place 1..N so that
// the sum of |preferred place - actual place| is least. Sort the wishes.
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;  // blank lines between cases are skipped by >>
    while (t--) {
        int n;
        cin >> n;
        vector<int> wish(n);
        string name;  // team names do not matter for the answer
        for (int& w : wish) cin >> name >> w;
        sort(wish.begin(), wish.end());
        long long badness = 0;  // up to about N*N/2, too big for int
        for (int i = 0; i < n; i++) badness += llabs((long long)wish[i] - (i + 1));
        cout << badness << '\n';
    }
    return 0;
}
```

The Python version, **Listing 10.8**, uses a different sort. Because every wish is an integer from 1 to N, a *counting sort* does the job in $O(N)$: count how many teams wish each place, then hand out the places in order of the wishes. The teams with the same wish take consecutive places, which is exactly what sorting would do.

Listing 10.8 `baised.py`: the same greedy with a counting sort.

```python
"""BAISED (Biased Standings): distinct places 1..N with the least sum of
|preferred - actual|. Sort the wishes (a counting sort, since 1 <= wish <= N)."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()  # blank lines vanish here
    t = int(data[0])
    pos, out = 1, []
    for _ in range(t):
        n = int(data[pos])
        count = [0] * (n + 2)
        for w in data[pos + 2 : pos + 2 + 2 * n : 2]:  # skip the team names
            count[int(w)] += 1
        pos += 1 + 2 * n
        place, badness = 1, 0
        for wish in range(1, n + 1):
            for _ in range(
                count[wish]
            ):  # teams with this wish take the next places
                badness += abs(wish - place)
                place += 1
        out.append(str(badness))
    print("\n".join(out))


main()
```

We use our own teams, named after birds. In the first test case seven teams wish for places 1,2,1,1,2,3,5; in the second, three teams all wish for first place.

```
$ printf '2\n\n7\nGaruda 1\nElang 2\nRajawali 1\nMerpati 1\nKutilang 2\nNuri 3\nKakatua
5\n\n3\nA 1\nB 1\nC 1\n' | ./baised
$ printf '2\n\n7\nGaruda 1\nElang 2\nRajawali 1\nMerpati 1\nKutilang 2\nNuri 3\nKakatua
5\n\n3\nA 1\nB 1\nC 1\n' | python3 ../python/baised.py
```

```
13
3
13
3
```

For the first case the sorted wishes are 1,1,1,2,2,3,5 against places 1, …,7, so by Equation (10.4) the badness is $0 + 1 + 2 + 2 + 3 + 3 + 2 = 13$. In the second, the three teams get places 1, 2 and 3: $0 + 1 + 2 = 3$.

10.7 Moving Goods Along a Line (GERGOVIA)

The problem in our words. In GERGOVIA (Wine trading in Gergovia) [11], N houses stand in a row, $2 \le N \le 10^5$, equally spaced. House i wants to buy a_i bottles of wine if $a_i > 0$ or to sell $-a_i$ bottles if $a_i < 0$, with $|a_i| \le 1000$, and the total demand equals the total supply: $a_1 + \cdots + a_N = 0$. Carrying one bottle from a house to its neighbor costs one unit of work. Find the least total work. Test cases follow until a line with 0.

Picture the same situation in a village: a row of stalls along a road, some with too much rice and some with too little, and a porter who is paid per sack per stall passed. The question is the least total porterage.

The key observation looks at the *borders* between neighboring houses instead of at the houses. Consider the border between house i and house $i + 1$, and let

$$P_i = a_1 + a_2 + \cdots + a_i$$

be the net demand of all houses to the left of it. If $P_i > 0$, the left side wants P_i more bottles than it has, and they can only come across this border from the right; if $P_i < 0$, the left side has $-P_i$ bottles too many, and they must cross to the right. Either way, at least $|P_i|$ bottles cross this border, and each crossing costs one unit. Summing over all borders gives a lower bound on the work:

$$work \ge \sum_{i=1}^{N-1} |P_i| \tag{10.5}$$

The greedy plan meets the bound. Walk from left to right carrying the running surplus or debt: at each border, pass the current P_i to the right (or bring $-P_i$ from the right, which is the same flow in the other direction). Each border is crossed by exactly $|P_i|$ bottles, never more, so the work is exactly the right-hand side of Equation (10.5). A lower bound that is achieved is the optimum. Here the “exchange argument” takes the form of a matching lower bound: any plan must pay at least $|P_i|$ at every border, and the greedy plan pays exactly that.

Since $P_N = 0$, summing $|P_i|$ up to N instead of $N - 1$ adds nothing, and the programs do that for simplicity. When the first half of the houses sell 1000 bottles each and the second half buy 1000

each, $|P_i|$ climbs to 5×10^7 and the work to about 2.5×10^{12}, far beyond `int`, so C++ uses `long long`.

Listing 10.9 `gergovia.cpp`: the sum of the absolute prefix sums.

```cpp
// GERGOVIA (Wine trading in Gergovia): houses on a line buy (+) or sell (-)
// wine; moving one bottle to a neighbor costs 1. Least total work: every
// border between house i and i+1 must carry |a[0] + ... + a[i]| bottles.
#include <cstdlib>
#include <iostream>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    while (cin >> n && n != 0) {
        long long carry = 0;  // bottles that must cross the current border
        long long work = 0;
        for (int i = 0; i < n; i++) {
            long long a;
            cin >> a;
            carry += a;
            work += llabs(carry);
        }
        cout << work << '\n';
    }
    return 0;
}
```

Listing 10.10 `gergovia.py`: the same sum with `itertools.accumulate`.

```python
"""GERGOVIA (Wine trading in Gergovia): the border after house i carries
|a[0] + ... + a[i]| bottles; the least total work is the sum of these."""

import sys
from itertools import accumulate


def main() -> None:
    data = sys.stdin.buffer.read().split()
    pos, out = 0, []
    while True:
        n = int(data[pos])
        if n == 0:
            break
        a = map(int, data[pos + 1 : pos + 1 + n])
        pos += 1 + n
        out.append(str(sum(abs(carry) for carry in accumulate(a))))
    print("\n".join(out))


main()
```

Listing 10.9 keeps the running prefix in `carry`; **Listing 10.10** lets `itertools.accumulate` produce all prefix sums at once. Our test uses two villages: four houses with $3, -1, -4, 2$ and six with $-2, 0, 5, -1, -3, 1$.

```
$ printf '4\n3 -1 -4 2\n6\n-2 0 5 -1 -3 1\n0\n' | ./gergovia
$ printf '4\n3 -1 -4 2\n6\n-2 0 5 -1 -3 1\n0\n' | python3 ../python/gergovia.py
```

```
7
10
7
10
```

For the first village the prefix sums are $3,2,-2,0$, so the work is $3+2+2=7$: three bottles cross the first border, two the second, two the third. Follow the porter to see that 7 is really achieved. House 1 wants 3 bottles and house 3 has 4 to sell; house 2 has 1 to sell and house 4 wants 2. At the border between houses 1 and 2, three bottles must travel left, because house 1 wants 3 and has none. House 2 supplies one of them, so two more must come across the border between houses 2 and 3, from house 3. House 3 has 4; it sends 2 to the left and must send the other 2 to house 4, across the third border. The bottles crossing the three borders are 3, 2 and 2, exactly the absolute prefix sums, and no plan can do with fewer, because each of those crossings is forced by Equation (10.5). For the second they are $-2,-2,3,2,-1,0$, and the work is $2+2+3+2+1=10$.

10.8 Testing It: Every Greedy Against a Brute Force

The proofs above are the real reason to trust these programs, but proofs can have gaps and programs can have typos. The stress driver of Section 9.10 is reused unchanged, with the four problems of this chapter, their random generators in `tests/cases.py` and the brute forces of **Table 10.1** in `tests/brute.py`. Each brute force tries *every* possibility, which is exactly what a greedy algorithm claims to make unnecessary.

Table 10.1 The brute forces used to check the four greedy programs.

Problem	Brute force	Small inputs
BUSYMAN	every subset of activities, kept if no two overlap	up to 8 activities in [0,12]
FASHION	every permutation of the women	up to 7 pairs
BAISED	every assignment of the places $1..N$	up to 7 teams
GERGOVIA	every way to send each bottle to a buyer	up to 6 houses, at most 2 bottles each

The GERGOVIA brute force deserves a word. It turns every bottle into a separate object, lists the position of each bottle to be sold and of each bottle to be bought, and tries every way to match sellers' bottles with buyers' bottles, adding the distances. It knows nothing about borders or prefix sums, so it is a genuinely independent check of Equation (10.5).

```
$ cd tests
$ python3 stress.py 300
```

```
busyman   300 random inputs: C++ and Python agree with brute force
fashion   300 random inputs: C++ and Python agree with brute force
baised    300 random inputs: C++ and Python agree with brute force
gergovia  300 random inputs: C++ and Python agree with brute force
```

All 1200 random inputs agree. It is instructive to break a program on purpose and watch the driver react: change `s >= freeAt` to `s > freeAt` in `busyman.cpp`, or sort by start time instead of end time, and run the driver again. Within a few rounds it prints a small failing input, usually two or three activities, together with the expected and the actual answer. That small input is the counterexample you would otherwise have searched for by hand, and it is exactly what a greedy rule without a proof deserves.

10.9 Making It Better: Speed in Both Languages

The greedy programs are all "sort, then one pass", so they are fast in C++. **Table 10.2** gives the measured CPU times on the largest inputs, produced by `tests/timing.py`.

```
program          C++ (s)  Python (s)  same output
busyman             0.14        0.73  yes
fashion             0.01        0.03  yes
baised              0.02        0.03  yes
gergovia            0.04        0.16  yes
```

Table 10.2 Largest inputs and measured CPU time for the four greedy programs.

Problem	Largest input	Work	C++ (s)	Python (s)
BUSYMAN	10 cases of 10^5 activities	sort, $O(N \log N)$	0.14	0.73
FASHION	100 cases of 1000 pairs	two sorts	0.01	0.03
BAISED	10^5 teams	sort (C++), counting sort (Python)	0.02	0.03
GERGOVIA	10 cases of 10^5 houses	one pass	0.04	0.16

BUSYMAN is the one to watch: its SPOJ time limit is tight (a fraction of a second [1]), and with up to a million activities in one file even the C++ program spends most of its time reading numbers and sorting. The fast input of Chapter 4 matters here, and Python, at 0.73 seconds on our machine, is unlikely to pass. Two improvements help C++ further when needed: read the input with a block reader such as the one in the EKO program of Chapter 11, and, because the times are integers up to 10^6, replace the comparison sort by a counting sort on the end times, which brings the work down to $O(N + 10^6)$.

For BAISED, the counting sort in **Listing 10.8** is the reason Python is as fast as C++ here: the wishes are small integers, so the sort costs $O(N)$ instead of $O(N \log N)$. Whenever the keys of a greedy sort are small integers, remember this option.

10.9.1 When any two piles may be merged

Chapter 9 solved MIXTURES, where only *neighboring* mixtures may be mixed, with an interval DP. Change one word and the problem becomes greedy. A rice mill receives sacks from n farmers and must pour them together into one heap, two heaps at a time; pouring heaps of a and b kilograms costs $a + b$ in labor, and *any* two heaps may be poured together. The cheapest plan is to pour the two *smallest* heaps together, every time. **Pseudocode 10.1** writes it down with a priority queue, which hands out the smallest element in $O(\log n)$.

Pseudocode 10.1 Merging heaps two at a time at the least total cost: always merge the two smallest.

```
MERGE-COST(a[1..n])
1  Q = priority queue containing a[1], ..., a[n]
2  total = 0
3  while Q holds more than one heap
4      x = EXTRACT-MIN(Q)
5      y = EXTRACT-MIN(Q)
6      total = total + x + y
7      INSERT(Q, x + y)
8  return total
```

By hand, for heaps of 4, 3, 2 and 6 kilograms: pour 2 and 3 (cost 5), then 4 and 5 (cost 9), then 6 and 9 (cost 15), in total 29. Pouring the two largest first, 4 and 6, and then adding the 2 and the 3 in turn, would cost $10 + 12 + 15 = 37$. The proof is again an exchange. Every plan is a binary tree whose leaves are the original heaps, and a heap's weight is paid once for every pour it takes part in, that is, once per level above its leaf. Take an optimal tree and look at its deepest pair of sibling leaves; swapping the two smallest heaps into those places cannot increase the cost, because it moves larger weights up and smaller weights down. So some optimal plan starts by pouring the

two smallest heaps, and the rest of the argument repeats on the remaining heaps. This is Huffman's algorithm, invented in 1952 to build the shortest prefix codes for compressing text [12], and it is still inside every ZIP file today. Placed next to MIXTURES, it shows the whole theme of this chapter in one picture: the same merging costs, one freedom removed, and the right tool switches from greedy to DP.

10.10 Using It for Real

Greedy thinking is everywhere in daily work, and so are wrong greedy rules. **Table 10.3** collects the questions to ask before trusting one.

Table 10.3 Questions that decide between a greedy algorithm and something stronger.

Question	If the answer is yes	Example
Can an exchange of two elements in any optimal answer turn it into the greedy answer without loss?	greedy is correct; write the proof down	BUSYMAN, FASHION, BAISED
Is there a lower bound that the greedy answer meets exactly?	greedy is correct	GERGOVIA
Does a brute force on thousands of small random inputs ever disagree?	greedy is wrong; keep the counterexample	coins 1, 3, 4; shortest first
Must each item be taken whole, with a budget (0/1)?	suspect DP, not greedy	KNAPSACK (Chapter 9)
Can items be divided (fractional)?	greedy by value per unit often works	fractional knapsack (Exercise 4)

A few everyday uses, now with the right rule:

- **Booking one room or one machine.** A clinic's single ultrasound machine, a *bengkel*'s single lift, Pak Harun's hall: to serve the most requests, take the request that *ends* first among those that fit. If the requests have different values (a wedding pays more than a meeting), the goal is no longer a count, and the problem becomes *weighted* interval scheduling, which needs DP [3].
- **Pairing to maximize total output.** Assign the strongest workers to the machines with the highest rate: sort both and pair in order, by Equation (10.3). To *minimize* a sum of products (for example, pairing heavy loads with short distances), sort one list ascending and the other descending.
- **Moving stock between shops on one road.** The least total transport is the sum of the absolute prefix sums, Equation (10.5), and the plan is to pass the surplus along the road.
- **Finishing orders on time.** A tailor with one sewing machine has orders with known sewing times and promised dates, and wants the *latest* order to be as little late as possible. Sew in order of promised date, *earliest deadline first*. The exchange argument looks at two neighboring orders A and B where A is sewn first but B has the earlier deadline. Swap them. The pair still finishes at the same moment T. After the swap, B finishes earlier than before, so its lateness can only shrink, and A now finishes at T, when B used to finish; because A's deadline is later than B's, A's lateness at T is smaller than B's lateness was at T. So the largest lateness cannot grow. Removing such *inversions* one at a time turns any schedule into the sorted one [3].
- **Giving change.** For rupiah, largest first is optimal. For any other set of denominations, check it with **Listing 10.1** before relying on it.

For more greedy problems with proofs, Kleinberg and Tardos [3] give the most careful treatment of exchange arguments, and Laaksonen [13] collects the greedy patterns most common in contests.

10.11 Common Mistakes, and How to Avoid Them

- **Choosing the key by intuition.** Earliest start and shortest first both sound right for activity selection and both are wrong. Test every candidate key against a brute force; the right one is usually the one you can prove.
- **Strict instead of non-strict comparison.** In BUSYMAN an activity may start when the previous one ends: the test is `s >= freeAt`, not `s > freeAt`. Read the statement for the words "may start at the same time".
- **Believing a greedy because it passes the samples.** Shortest first fails on only 1.6% of random small cases; sample tests rarely contain the counterexample. Stress test.
- **Using greedy on a 0/1 problem.** Value per unit is the right rule when items can be cut, and a wrong one when they cannot. The knapsack of Chapter 9 needs DP.
- **Overflow in sums.** BAISED can reach about 5×10^9 and GERGOVIA about 2.5×10^{12}. Use `long long`.
- **Tripping over the input format.** Blank lines between test cases and names before numbers are harmless with token-based reading, and deadly with line-based reading that expects numbers on every line.
- **Sorting a copy and using the original.** When a greedy needs the elements in a new order, make sure every later step uses the sorted order, or keep the original indices in pairs.

10.12 Chapter Summary

- A greedy algorithm makes the locally best choice at every step and never takes it back. It is fast and short, and it is correct only when the problem's structure allows it.
- An exchange argument proves a greedy rule: swap an optimal solution, step by step, into the greedy one without making it worse. For activity selection the key step is Equation (10.1); for pairing, the uncrossing gain Equation (10.3); for ranking, the sorted assignment Equation (10.4); for transport, the matching lower bound Equation (10.5).
- A single counterexample disproves a greedy rule, and a brute force on random small inputs finds one fast: **Listing 10.2** rejected two of three plausible rules, and **Listing 10.1** showed that largest-coin-first is right for rupiah but wrong for coins 1, 3, 4.
- BUSYMAN, FASHION, BAISED and GERGOVIA each became a sort and a loop in C++17 and Python 3, checked on 1200 random inputs against brute forces. C++ is fast everywhere; Python is fine except for BUSYMAN's tight limit.

10.13 Exercises

1. Run the earliest-end rule by hand on the activities [1,4), [3,5), [0,6), [5,7), [3,9), [5,9), [6,10), [8,11), [8,12), [2,14), [12,16). Which activities are taken, and how many?
2. Is the rule "latest start first" (take the activity that starts last, then the one that starts last among those ending in time, and so on, working backward through the day) correct for BUSYMAN? Prove it or give a counterexample.
3. A coin system has coins 1, 5 and 12. Find the smallest amount for which greedy change-making is not optimal, and give both answers.
4. In the *fractional knapsack*, goods can be cut, and a piece of weight x of a good with weight w and value v is worth $x \cdot v/w$. Show with an exchange argument that taking goods in order of value

per kilogram is optimal, and use **Listing 10.11** to compare the fractional and the 0/1 answers for three goods of 10, 20 and 30 kg worth 60, 100 and 120, with a capacity of 50 kg.

5. For FASHION, what rule gives the *smallest* possible sum of products? Prove it with Equation (10.3).
6. In GERGOVIA, why does it not matter in which order the actual bottles are carried? Explain with Equation (10.5).

10.14 Solutions to the Exercises

1. Sorted by end time: [1,4), [3,5), [0,6), [5,7), [3,9), [5,9), [6,10), [8,11), [8,12), [2,14), [12,16). Take [1,4), free at 4. [3,5) and [0,6) start too early. Take [5,7), free at 7. [3,9), [5,9) and [6,10) start before 7. Take [8,11), free at 11. [8,12) and [2,14) start too early. Take [12,16). Four activities: [1,4), [5,7), [8,11), [12,16). The program agrees:

```
$ printf '1\n11\n1 4\n3 5\n0 6\n5 7\n3 9\n5 9\n6 10\n8 11\n8 12\n2 14\n12 16\n' |
./busyman
```

```
4
```

2. Yes. Reverse time: replace every activity $[s, e)$ by $[-e, -s)$. Two activities overlap before the reversal exactly when they overlap after it, so the best count does not change. In the reversed day, "latest start first" becomes "earliest end first", which Section 10.4.2 proved optimal. The exchange argument can also be written directly: the activity that starts last can replace the last activity of any optimal schedule, because it starts no earlier, so it cannot overlap the one before it.

3. Amount 15: greedy takes $12 + 1 + 1 + 1$, four coins, while $5 + 5 + 5$ uses three. Smaller amounts are all fine: below 12 greedy uses only 5s and 1s, which is optimal, and for 12, 13 and 14 it uses 12 plus one, two or three 1s, which cannot be beaten, since without the 12 even 14 needs $5 + 5 + 1 + 1 + 1 + 1$. Changing `compare` in **Listing 10.1** to the coins {1,5,12} confirms 15 as the first failure.

4. Suppose an optimal load contains less of the good with the highest value per kilogram, r_1, than greedy takes, while it contains some weight of a good with a lower rate $r_2 < r_1$. Exchange δ kilograms of the second good for δ kilograms of the first: the weight is unchanged and the value rises by $\delta(r_1 - r_2) > 0$, a contradiction. So an optimal load takes as much as possible of the best-rate good, then of the next, which is the greedy rule. **Listing 10.11** shows the numbers:

Listing 10.11 `fractional.cpp`: the fractional knapsack by value per kilogram, next to the 0/1 answer found by brute force.

```cpp
// Exercise 10.4: fractional knapsack (goods may be cut) solved greedily by
// value per kilogram, next to 0/1 knapsack on the same goods, where it fails.
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

struct Good {
    double kg, value;
};

int main() {
    vector<Good> g = {{10, 60}, {20, 100}, {30, 120}};
    double cap = 50;
    sort(g.begin(), g.end(),
         [](Good a, Good b) { return a.value / a.kg > b.value / b.kg; });
    double room = cap, total = 0;
    for (Good x : g) {
        double take = min(room, x.kg);  // as much of the best-per-kg good as fits
        total += take / x.kg * x.value;
        room -= take;
        printf("take %4.1f kg of the %2.0f kg good\n", take, x.kg);
    }
    printf("fractional (greedy by value per kg): %.1f\n", total);
    // 0/1 version: the same greedy takes 10 and 20 kg (value 160) and stops.
    double best = 0;
    for (int mask = 0; mask < 8; mask++) {
        double kg = 0, value = 0;
        for (int i = 0; i < 3; i++)
            if (mask >> i & 1) kg += g[i].kg, value += g[i].value;
        if (kg <= cap) best = max(best, value);
    }
    printf("0/1 (brute force over 8 subsets):    %.1f\n", best);
    return 0;
}
```

```
$ ./fractional
```

```
take 10.0 kg of the 10 kg good
take 20.0 kg of the 20 kg good
take 20.0 kg of the 30 kg good
fractional (greedy by value per kg): 240.0
0/1 (brute force over 8 subsets):    220.0
```

The fractional answer is 240: all of the goods with rates 6 and 5 per kilogram, and 20 of the 30 kg with rate 4. For whole goods the best is 220 (20 kg and 30 kg), and the greedy order would have taken 10 kg and 20 kg for only 160 before running out of room for the third. Exchange arguments need the freedom to exchange δ kilograms; whole goods do not allow it, which is exactly why the $0/1$ knapsack needs DP.

5. Sort one list ascending and the other descending, pairing the largest man with the lowest-rated woman. By Equation (10.3), uncrossing two pairs *raises* the sum by $(a_2 - a_1)(b_2 - b_1) \geq 0$, so *crossing* them lowers it by the same amount. Starting from any pairing and crossing every uncrossed pair never increases the sum and ends at the fully crossed, opposite-order pairing, which is therefore the minimum.

6. Equation (10.5) depends only on how many bottles cross each border, not on which bottles they are or in which order they travel. Any plan in which exactly $|P_i|$ bottles cross border i, all in the same direction, costs exactly the bound. Plans that send bottles both ways across a border, or more bottles than needed, pay more, and those are the only plans that are not optimal.

References

[1] Sphere Online Judge, "BUSYMAN - I AM VERY BUSY." [Online]. Available: https://www.spoj.com/problems/BUSYMAN/

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 15.

[3] J. Kleinberg and É. Tardos, *Algorithm Design*. Boston, MA, USA: Pearson/Addison-Wesley, 2006, ch. 4 and 6.

[4] Bank Indonesia, "Bank Indonesia dan Pemerintah Meluncurkan Uang Rupiah Kertas Tahun Emisi 2022," press release, Aug. 18, 2022. [Online]. Available: https://www.bi.go.id/id/publikasi/ruang-media/news-release/Pages/sp_2421922.aspx

[5] Kompas.com, "Daftar Uang Rupiah Ini Masih Berlaku, Tak Boleh Ditolak Saat Transaksi," Oct. 8, 2024. [Online]. Available: https://www.kompas.com/tren/read/2024/10/08/183000765/daftar-uang-rupiah-ini-masih-berlaku-tak-boleh-ditolak-saat-transaksi

[6] D. Kozen and S. Zaks, "Optimal bounds for the change-making problem," *Theoretical Computer Science*, vol. 123, no. 2, pp. 377–388, 1994.

[7] D. Pearson, "A polynomial-time algorithm for the change-making problem," *Operations Research Letters*, vol. 33, no. 3, pp. 231–234, 2005.

[8] Sphere Online Judge, "FASHION - Fashion Shows." [Online]. Available: https://www.spoj.com/problems/FASHION/

[9] G. H. Hardy, J. E. Littlewood, and G. Pólya, *Inequalities*, 2nd ed. Cambridge, U.K.: Cambridge University Press, 1952, Sec. 10.2.

[10] Sphere Online Judge, "BAISED - Biased Standings." [Online]. Available: https://www.spoj.com/problems/BAISED/

[11] Sphere Online Judge, "GERGOVIA - Wine trading in Gergovia." [Online]. Available: https://www.spoj.com/problems/GERGOVIA/

[12] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[13] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

Chapter 11

Binary Search on the Answer and Two Pointers

Some optimization problems are hard to answer directly but easy to *check*: “what is the largest possible distance?” is hard, while “is a distance of 300 possible?” can be answered with one pass over the data. When the answers to the easy question switch from yes to no only once as the number grows, a binary search over the number finds the best value with a handful of checks. This technique, *binary search on the answer*, turns many “maximize the minimum” and “minimize the maximum” problems into short programs. Its cousin, the *two-pointer* technique, walks two indices through an array in the same direction and replaces a quadratic search over all subarrays by a linear one. By the end of this chapter you will be able to recognize a monotone yes/no question inside a problem, write a binary search that never loops forever and never misses the boundary, do the same on real numbers, rebuild the actual solution once the best value is known, and solve five SPOJ problems in C++17 and Python 3: AGGRCOW, EKO, PIE, BOOKS1 and HOTELS.

11.1 Bu Ratna’s Night Market

Every Saturday evening Bu Ratna sets up the night market along the main road of her town. The road has a row of spots where a stall can stand, at known distances from the mosque at the end of the road. This week four *martabak* sellers have registered, and every one of them has asked the same thing: “Please don’t put me right next to another martabak stall.” Bu Ratna wants to be fair. She decides to place the four martabak stalls so that the *smallest* distance between any two of them is as large as possible.

With twelve possible spots, she could try every choice of four spots, but that is 495 choices for four stalls and far too many for a real market with a hundred spots and twenty sellers. There is a better way to think. Instead of asking “what is the best smallest distance?”, Bu Ratna asks a simpler question: “can I place the four stalls so that every two are at least 300 meters apart?” That question is easy. Walk along the road, put the first stall at the first spot, and put each next stall at the first spot that is at least 300 meters beyond the previous one. If four stalls fit, the answer is yes.

And the easy question has a useful shape: if 300 meters is possible, then 250 meters is certainly possible too, and if 320 meters is impossible, then 400 meters is impossible as well. The answers form a block of yes followed by a block of no. Finding where the block of yes ends is a job for binary search.

Bu Ratna’s question is the SPOJ problem AGGRCOW (Aggressive cows) [1], with stalls for cows instead of spots for stalls. It opens this chapter, and the same idea then solves a sawmill problem, a birthday-cake problem and a copying-office problem.

11.2 The Idea in Plain Words

11.2.1 A question that changes its answer once

Let $ok(d)$ be the yes/no question "can the stalls be placed with every two at least d apart?". If $ok(d)$ is true, then $ok(d')$ is true for every smaller d', because a placement that keeps distance d keeps every smaller distance too. A question with this property is called *monotone*.

> **DEFINITION.** A yes/no question $ok(x)$ about a number x is *monotone* if its answers, for x increasing, are a block of *yes* followed by a block of *no* (or the other way round). *Binary search on the answer* finds the boundary between the two blocks by keeping two values, lo with $ok(lo)$ true and hi with $ok(hi)$ false, and repeatedly asking the question in the middle [2].

The search keeps one promise, its *invariant*, from start to finish:

$$ok(lo) = \text{yes}, \qquad ok(hi) = \text{no}, \qquad lo < hi \tag{11.1}$$

Each step picks mid strictly between lo and hi and asks $ok(mid)$. If the answer is yes, mid becomes the new lo; if no, it becomes the new hi. Either way Equation (11.1) still holds and the gap $hi - lo$ is halved. When the gap is 1, the two values are neighbors, so lo is the *largest* yes and hi the *smallest* no. The number of questions asked is

$$\lceil \log_2(hi_0 - lo_0) \rceil \tag{11.2}$$

where lo_0 and hi_0 are the starting values. For a range of 10^9 that is 30 questions; for 10^{18}, 60.

Three details make this template safe, and each prevents a classic bug:

- **Start with a real yes and a real no.** Choose lo_0 so that the answer is certainly yes (for AGGRCOW, distance 0 always works) and hi_0 so that it is certainly no (a distance larger than the whole road). Then the loop never has to guess what happens at the ends.
- **Stop when the gap is 1**, with the test `hi - lo > 1`. Because mid always lies strictly between the two, each step shrinks the gap, and the loop cannot run forever, a common failure of versions with `lo <= hi` and `mid + 1` or `mid - 1` placed wrongly.
- **Compute the middle as `lo + (hi - lo) / 2`.** The more obvious `(lo + hi) / 2` can overflow when both are large, a bug that sat unnoticed for years in widely used library code [3].

> **KEY INSIGHT:** Binary search on the answer changes the question. Instead of computing the best value, you only have to *check* a candidate value. Checking is usually a simple greedy pass; the binary search contributes a factor of about 30.

11.2.2 The same template in a sorted array

The classic binary search, looking for a value in a sorted array, is the same template with a particular question. For a sorted array $a_0 \le a_1 \le \cdots \le a_{n-1}$ and a value x, ask of every index i: "is $a_i \ge x$?". Because the array is sorted, the answers are a block of no followed by a block of yes, and the first yes is the position where x is, or where it would be inserted. That position is what `std::lower_bound` in C++ and `bisect.bisect_left` in Python return.

Work it by hand on $a = (3,8,8,12,20)$ with $x = 8$. The imaginary index -1 counts as no and the imaginary index 5 as yes, so start with $lo = -1$ and $hi = 5$. The middle is $-1 + \lfloor 6/2 \rfloor = 2$, and $a_2 = 8 \ge 8$, yes, so $hi = 2$. Next $mid = -1 + \lfloor 3/2 \rfloor = 0$, and $a_0 = 3 \ge 8$ is no, so $lo = 0$. Next $mid = 1$, and $a_1 = 8$, yes, so $hi = 1$. The gap is 1 and the answer is index 1, the first 8. With $x = 10$ the same steps end at index 3, where 10 would be inserted between 8 and 12. Counting how

many times a value occurs is then two searches: the first element $\geq x$ and the first element $>x$, whose difference is the count. The only thing that changed from the AGGRCOW search is the question; the loop, the invariant Equation (11.1) and the number of steps Equation (11.2) are the same.

11.2.3 When no upper end is known

Sometimes a problem gives no convenient bound for hi_0, or the bound is so large that most of the range is obviously useless. Then find the upper end by *doubling*: ask $ok(1)$, $ok(2)$, $ok(4)$, $ok(8)$, and so on, until the first no. If the true answer is A, doubling stops after about $\log_2 A$ questions at a value below $2A$, and the binary search between the last yes and the first no takes about $\log_2 A$ more. This *exponential search* costs about $2\log_2 A$ questions in total and never needs to know the size of the range in advance. It is also the safe choice when the check becomes expensive for large values, for example when checking a candidate means simulating that many days.

11.3 A First Attempt: Try Every Value

The straightforward way to find the largest yes is to try the candidates one by one: start at 1 and step up while the next value still works. **Listing 11.1** does that for Bu Ratna's market, twelve spots between 1 and 1000 meters and four martabak stalls, and then does the same job with binary search. It counts the calls of the check.

Listing 11.1 `bsearch_demo.cpp`: the check for a distance, a linear scan over all distances and a binary search, with the checks counted.

```
// A yes/no question that changes its answer only once, and two ways to find
// the change: try every value, or halve the range each time.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

vector<long long> pos = {1, 40, 95, 180, 260, 333, 410, 505, 590, 700, 810, 1000};
const int COWS = 4;
int checks = 0;

bool canPlace(long long gap, int cows) {  // can every two cows be >= gap apart?
    checks++;
    int placed = 1;
    long long last = pos[0];
    for (size_t i = 1; i < pos.size(); i++)
        if (pos[i] - last >= gap)
            placed++, last = pos[i];  // first stall far enough
    return placed >= cows;
}

int main() {
    sort(pos.begin(), pos.end());
    cout << "gap   ";
    for (int g = 240; g <= 340; g += 20) cout << g << ' ';
    cout << "\nfits? ";
    for (int g = 240; g <= 340; g += 20)
        cout << (canPlace(g, COWS) ? "yes " : "no  ");
    cout << '\n';

    checks = 0;
    long long g = 1;
    while (canPlace(g + 1, COWS)) g++;  // walk up while the next gap still works
    cout << "linear scan:   largest gap " << g << " after " << checks
         << " checks\n";
```

Listing 11.1 (Continued) *bsearch_demo.cpp: the check for a distance, a linear scan over all distances and a binary search, with the checks counted.*

```
    checks = 0;
    long long lo = 0,
              hi = pos.back() - pos.front() + 1;  // canPlace(lo), !canPlace(hi)
    while (hi - lo > 1) {
        long long mid = lo + (hi - lo) / 2;
        if (canPlace(mid, COWS))
            lo = mid;
        else
            hi = mid;
    }
    cout << "binary search: largest gap " << lo << " after " << checks
         << " checks\n";
    return 0;
}
```

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o bsearch_demo bsearch_demo.cpp
$ ./bsearch_demo
```

```
gap   240 260 280 300 320 340
fits? yes yes yes yes no  no
linear scan:   largest gap 300 after 300 checks
binary search: largest gap 300 after 10 checks
```

The first two lines show the monotone shape: yes up to 300, no from 320 on. The linear scan needs 300 checks to find the boundary; binary search needs 10, exactly $\lceil \log_2 1000 \rceil$ by

Equation (11.2). With positions up to 10^9, the scan would need up to a billion checks, each one a pass over 10^5 stalls; binary search still needs about 30 passes.

Figure 11.1 shows the first four questions of the binary search. The first, at 500, is no; the second, at 250, is yes; then 375 and 312 are both no. After four questions the answer is known to lie between 250 and 312, and six more questions (281, 296, 304, 300, 302, 301) close the gap to $lo = 300$, $hi = 301$.

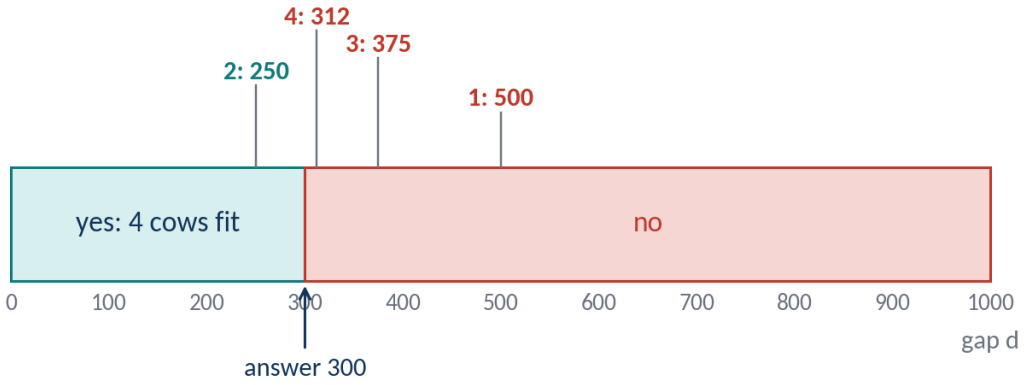


Figure 11.1 The answers to "can four stalls keep distance d?" form a block of yes and a block of no; the numbered marks are the first four questions of the binary search.

11.4 Maximizing the Minimum (AGGRCOW)

The problem in our words. AGGRCOW [1] has several test cases. Each gives N stall positions on a line, $2 \le N \le 10^5$, each between 0 and 10^9, and a number C of cows, $2 \le C \le N$. Put the cows in different stalls so that the smallest distance between two cows is as large as possible, and print that distance.

The check. For a distance d, sort the stalls and walk from left to right: the first cow goes into the first stall, and each next cow into the first stall at least d beyond the previous cow. If all C cows fit, $ok(d)$ is yes.

Why is placing each cow *as far left as possible* safe? This is a small exchange argument in the style of Chapter 10. Take any valid placement with distance d and compare it with the greedy one cow by cow. The greedy first cow is at the leftmost stall, no further right than the other placement's first cow. If the greedy j-th cow is no further right than the other's j-th cow, then the other's $(j+1)$-th cow, which is at least d beyond its j-th, is also at least d beyond the greedy j-th cow, so a stall that far away exists, and the greedy $(j+1)$-th cow is placed no further right. By induction the greedy places every cow the other placement places, so if any placement exists, the greedy finds one.

The search. Distance 0 always works, so $lo_0 = 0$. A distance larger than the span of the stalls never works, so $hi_0 = x_{\max} - x_{\min} + 1$. Sorting costs $O(N \log N)$ and each check $O(N)$, so the total is $O(N \log N + N \log X)$, where $X \le 10^9$ is the span.

Listing 11.2 `aggrcow.cpp`: binary search on the distance with the greedy check.

```cpp
// AGGRCOW (Aggressive cows): put C cows in N stalls on a line so that the
// smallest distance between two cows is as large as possible.
// Binary search on the answer d with a greedy yes/no check.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

bool fits(const vector<long long>& x, int cows, long long d) {
    int placed = 1;  // the first cow always goes to the leftmost stall
    long long last = x[0];
    for (size_t i = 1; i < x.size() && placed < cows; i++)
        if (x[i] - last >= d) {
            placed++;
            last = x[i];
        }
    return placed >= cows;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;
    while (t--) {
        int n, c;
        cin >> n >> c;
        vector<long long> x(n);
        for (auto& v : x) cin >> v;
        sort(x.begin(), x.end());
        long long lo = 0,
                  hi = x[n - 1] - x[0] + 1;  // fits(lo) is true, fits(hi) false
        while (hi - lo > 1) {
            long long mid = lo + (hi - lo) / 2;
            if (fits(x, c, mid))
                lo = mid;
            else
                hi = mid;
        }
        cout << lo << '\n';
    }
    return 0;
}
```

Listing 11.3 `aggrcow.py`: the same search in Python, with an early exit once all cows fit.

```python
"""AGGRCOW (Aggressive cows): make the smallest distance between two of the
C cows as large as possible. Binary search on the answer, greedy check."""

import sys


def fits(x: list[int], cows: int, d: int) -> bool:
    placed, last = 1, x[0]
    for v in x:
        if v - last >= d:  # the first stall far enough from the last cow
            placed += 1
            last = v
            if placed == cows:
                return True
    return placed >= cows


def main() -> None:
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    pos, out = 1, []
    for _ in range(t):
        n, c = int(data[pos]), int(data[pos + 1])
        x = sorted(map(int, data[pos + 2 : pos + 2 + n]))
        pos += 2 + n
        lo, hi = 0, x[-1] - x[0] + 1  # fits(lo) is true, fits(hi) is false
        while hi - lo > 1:
            mid = (lo + hi) // 2
            if fits(x, c, mid):
                lo = mid
            else:
                hi = mid
        out.append(str(lo))
    print("\n".join(out))


main()
```

Listing 11.2 stores positions in `long long` so that differences are safe, and stops the check as soon as all cows are placed. **Listing 11.3** returns from `fits` at the same moment, which matters in Python: with many cows placed early, most of the stalls are never looked at. Our test uses Bu Ratna's twelve spots with four sellers and a second, smaller road with five stalls and three cows.

```
$ printf '2\n12 4\n1\n40\n95\n180\n260\n333\n410\n505\n590\n700\n810\n1000\n5
3\n10\n1\n7\n3\n12\n' | ./aggrcow
$ printf '2\n12 4\n1\n40\n95\n180\n260\n333\n410\n505\n590\n700\n810\n1000\n5
3\n10\n1\n7\n3\n12\n' | python3 ../python/aggrcow.py
```

```
300
5
300
5
```

For the second road, sorted as 1,3,7,10,12, a distance of 5 places cows at 1, 7 and 12, while 6 places only two (1 and 7, and nothing is 6 beyond 7). The answer is 5.

11.5 The Highest Saw (EKO)

The problem in our words. In EKO [4], a woodcutter needs at least M meters of wood, $M \le 2 \times 10^9$. There are $N \le 10^6$ trees in a row with heights below 10^9. His machine raises a saw to a height H and cuts every tree above it; he takes the parts above H. Being careful with the forest, he

wants H as *high* as possible while still getting at least M meters. The total height of all trees is more than M. Print the best H.

The monotone question is $ok(H)$: "does cutting at height H give at least M meters?". The wood collected,

$$wood(H) = \sum_{i=1}^{N} \max(0,\ h_i - H) \tag{11.3}$$

can only shrink as H grows, so the answers are yes up to some height and no above it: the same shape as AGGRCOW. $H = 0$ gives all the wood, more than M, so $lo_0 = 0$; at the height of the tallest tree nothing is cut, so $hi_0 = h_{\max}$.

Pitfalls. Two traps are waiting. First, Equation (11.3) can reach $10^6 \times 10^9 = 10^{15}$, so the sum must be `long long`; with `int` it overflows, wraps to a negative number, and the search goes the wrong way. Second, reading a million numbers is a real cost against the problem's tight time limit, so **Listing 11.4** reads the input in blocks of 64 KB with `fread` and parses the digits itself, the technique of Chapter 4. With 30 checks of a million trees, the whole search is about 3×10^7 simple steps.

Listing 11.4 `eko.cpp`: binary search on the saw height, with block input and a 64-bit sum.

```cpp
// EKO: set the saw height H as high as possible while the parts of the trees
// above H still give at least M meters of wood. Binary search on H.
// A million numbers arrive, so the input is read in large blocks (Chapter 4).
#include <cstdio>
#include <vector>
using namespace std;

static char buf[1 << 16];
size_t len = 0, at = 0;

int readChar() {
    if (at == len) {
        len = fread(buf, 1, sizeof buf, stdin);
        at = 0;
        if (len == 0) return -1;
    }
    return buf[at++];
}

long long readInt() {
    int ch = readChar();
    while (ch != -1 && (ch < '0' || ch > '9')) ch = readChar();
    long long v = 0;
    for (; ch >= '0' && ch <= '9'; ch = readChar()) v = v * 10 + (ch - '0');
    return v;
}

int main() {
    int n = readInt();
    long long m = readInt();
    vector<long long> h(n);
    long long top = 0;
    for (auto& v : h) {
        v = readInt();
        if (v > top) top = v;
    }
    long long lo = 0, hi = top;  // height lo gives enough wood, height hi does not
    while (hi - lo > 1) {
        long long mid = lo + (hi - lo) / 2, wood = 0;
        for (long long v : h)
            if (v > mid) wood += v - mid;  // can reach 10^15: needs long long
        if (wood >= m)
            lo = mid;
        else
            hi = mid;
    }
```

Listing 11.4 (Continued) *eko.cpp: binary search on the saw height, with block input and a 64-bit sum.*

```cpp
    printf("%lld\n", lo);
    return 0;
}
```

A different Python solution. A Python loop over a million trees takes a noticeable fraction of a second, and 30 of them are far too slow. **Listing 11.5** therefore avoids the binary search. It sorts the heights from tallest to shortest, $h_1 \geq h_2 \geq \cdots \geq h_N$, and adds $h_{N+1} = 0$. When the saw is between h_{k+1} and h_k, exactly the k tallest trees are cut, and with $T_k = h_1 + \cdots + h_k$ the wood is $T_k - kH$. The largest H in that stretch that still gives M meters is

$$H = \left\lfloor \frac{T_k - M}{k} \right\rfloor \tag{11.4}$$

provided it is at least h_{k+1}. Walking $k = 1,2,\ldots$ until that happens gives the answer in one pass after the sort.

Listing 11.5 `eko.py`: EKO without binary search, using Equation (11.4) on the sorted heights.

```
"""EKO: the highest saw height H that still yields at least M meters.
Python version: sort the trees once and walk down from the tallest, so no
binary search is needed (a binary search would read the million trees 30 times)."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    n, m = int(data[0]), int(data[1])
    h = sorted(map(int, data[2 : 2 + n]), reverse=True)
    h.append(0)  # the ground: cutting at 0 takes everything
    total = 0  # sum of the k tallest trees
    for k in range(1, n + 1):
        total += h[k - 1]
        # with the saw between h[k] and h[k-1], only the k tallest trees are cut:
        # wood(H) = total - k*H, and we need wood(H) >= m
        best = (total - m) // k
        if best >= h[k]:  # the best height lies in this stretch
            print(min(best, h[k - 1]))
            return


main()
```

Both are tested on six trees of heights 30, 12, 25, 18, 40 and 9, with 25 meters needed.

```
$ printf '6 25\n30 12 25 18 40 9\n' | ./eko
$ printf '6 25\n30 12 25 18 40 9\n' | python3 ../python/eko.py
```

```
23
23
```

At $H = 23$ the three tallest trees give $17 + 7 + 2 = 26$ meters, enough; at $H = 24$ they give $16 + 6 + 1 = 23$, too little. By Equation (11.4) with $k = 3$ and $T_3 = 40 + 30 + 25 = 95$: $\lfloor (95 - 25)/3 \rfloor = 23$, and $23 \geq h_4 = 18$, so the walk stops there.

11.6 Binary Search on Real Numbers (PIE)

The problem in our words. In PIE [5], a host has N round pies of height 1 with integer radii up to 10^4, and F friends, $N, F \leq 10^4$. Everyone, the F friends and the host, must get a piece of the same volume, and each piece must come from a single pie (pieces may not be glued together; leftovers are fine). Print the largest possible volume, with an absolute error of at most 10^{-3}.

The monotone question: "can $F + 1$ pieces of volume V be cut?". A pie of volume $v_i = \pi r_i^2$ yields $\lfloor v_i / V \rfloor$ pieces, so

$$ok(V) \Leftrightarrow \sum_{i=1}^{N} \left\lfloor \frac{v_i}{V} \right\rfloor \geq F + 1 \tag{11.5}$$

Larger pieces mean fewer pieces, so the answers are yes up to the best V and no above it.

What changes with real numbers. There is no "gap of 1" to stop at, and comparing `hi - lo > 1e-9` can loop forever when rounding makes the midpoint equal to one of the ends. The safe way is a *fixed number of rounds*. The range starts at the volume of the largest pie, below $\pi \cdot 10^8 \approx$

3.1×10^8, and each round halves it; after t rounds the error is below $3.1 \times 10^8 / 2^t$, which is under 10^{-3} once $t \geq 39$. The C++ program uses 100 rounds and the Python program 60, both far more than needed and both cheap. Two more details: compute π as `acos(-1.0)`, which is exact to the precision of `double`, rather than typing 3.14159; and use `double`, never `float`, whose 24-bit precision cannot hold volumes near 10^8 to three decimals [6].

Listing 11.6 `pie.cpp`: binary search on a real number with a fixed number of rounds.

```cpp
// PIE: cut N round pies of height 1 into F+1 equal pieces, each piece from a
// single pie, as large as possible. Binary search on a real number.
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    const double PI = acos(-1.0);  // exact to double precision
    int t;
    if (!(cin >> t)) return 0;
    while (t--) {
        int n, f;
        cin >> n >> f;
        vector<double> vol(n);
        for (double& v : vol) {
            long long r;
            cin >> r;
            v = PI * r * r;
        }
        double lo = 0, hi = *max_element(vol.begin(), vol.end());
        for (int it = 0; it < 100; it++) {  // fixed rounds instead of hi - lo > eps
            double mid = (lo + hi) / 2;
            long long pieces = 0;
            for (double v : vol) pieces += (long long)(v / mid);
            if (pieces >= f + 1)
                lo = mid;  // f friends plus the host
            else
                hi = mid;
        }
        printf("%.4f\n", lo);
    }
    return 0;
}
```

Listing 11.7 `pie.py`: the same search in Python.

```python
"""PIE: the largest equal piece for F+1 people, each piece from one pie.
Binary search on a real number with a fixed number of rounds."""

import math
import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    pos, out = 1, []
    for _ in range(t):
        n, f = int(data[pos]), int(data[pos + 1])
        vol = [math.pi * int(r) ** 2 for r in data[pos + 2 : pos + 2 + n]]
        pos += 2 + n
        lo, hi = 0.0, max(vol)
        for _ in range(60):  # 2**-60 of the range is far below 1e-3
            mid = (lo + hi) / 2
            if sum(int(v / mid) for v in vol) >= f + 1:
                lo = mid
            else:
                hi = mid
        out.append(f"{lo:.4f}")
    print("\n".join(out))


main()
```

Listing 11.6 counts pieces in `long long`, because a tiny candidate volume at the start of the search makes $\lfloor v_i/V \rfloor$ enormous; **Listing 11.7** relies on Python's unbounded integers. We try two parties: pies of radius 5 and 3 for five friends, and pies of radius 1 to 4 for ten friends.

```
$ printf '2\n2 5\n5 3\n4 10\n1 2 3 4\n' | ./pie
$ printf '2\n2 5\n5 3\n4 10\n1 2 3 4\n' | python3 ../python/pie.py
```

```
15.7080
7.1808
15.7080
7.1808
```

In the first party six people share pies of volume $25\pi \approx 78.54$ and $9\pi \approx 28.27$. The answer $5\pi \approx 15.708$ cuts the big pie into exactly 5 pieces and the small one into 1, which is 6. In the second, eleven people are served by pieces of $16\pi/7 \approx 7.1808$: the largest pie gives 7 pieces, the pie of volume 9π gives 3, the pie of 4π gives 1, and the smallest gives none. In both cases the answer is some pie cut into equal parts, which is also how the brute force of Section 11.9 finds it.

11.7 Minimizing the Maximum, and Rebuilding the Answer (BOOKS1)

The problem in our words. In BOOKS1 (Copying Books) [7], m books with known page counts stand on a shelf in a fixed order, $m \le 500$, each with fewer than 10^7 pages, and $k \le m$ scribes will copy them. Each scribe gets a run of consecutive books, at least one. The work of a scribe is the total number of pages; the job is finished when the busiest scribe is finished. Split the books so that the largest work is as small as possible. If several splits achieve it, choose the one that gives the first scribe as little as possible, then the second, and so on. Print the books in order with `/` between the scribes' runs. There are about 200 test cases.

The question now is "minimize the maximum", the mirror image of AGGRCOW, and the output is not a number but the split itself. Both points are new.

The monotone question. For a cap X, can the books be split into at most k runs with no run above X pages? The check is again greedy: walk from the left, put books into the current run while it stays within X, and start a new run when the next book would overflow it. That greedy uses the fewest runs possible, by the same "stays ahead" argument as in AGGRCOW: each greedy run ends no earlier than the corresponding run of any other valid split. So

$$ok(X) \Leftrightarrow runs(X) \leq k \tag{11.6}$$

where $runs(X)$ is the number of runs the greedy needs. A larger cap never needs more runs, so the answers are no for small X and yes from the best cap on. This time we want the *smallest* yes, so the invariant is mirrored: lo is always a no, hi always a yes, and the answer is hi at the end. No cap below the thickest book can work, so lo_0 is one less than the thickest book; the sum of all pages always works, so hi_0 is that sum. (If there are fewer runs than scribes, some runs can be split further, since each run has at least one book and $k \leq m$; a run split in two never exceeds the cap.)

Rebuilding the split. Knowing the best cap X^* is not enough; we must print the split with the tie-breaking rule. Giving the first scribe as little as possible means giving the *later* scribes as much as possible. So the programs fill the runs from the *right*: walking from the last book to the first, they add books to the current run while it stays within X^*, and close the run (put a slash before it) when the next book would overflow it, or when the books still to the left are just enough to give each remaining scribe one book. The second condition is what guarantees that every scribe gets at least one book. Filling from the right makes the last scribe's run as long as possible, which leaves the fewest pages for the others; applied scribe by scribe, it minimizes the first scribe's work, then the second's, and so on. Because this tie-breaking argument is subtle, the stress test of Section 11.9 compares the printed split with a brute force that tries every split and applies the rule literally.

Listing 11.8 `books1.cpp`: binary search for the smallest cap, then the split rebuilt from the right.

```cpp
// BOOKS1 (Copying Books): split m books, in order, among k scribes so that the
// largest share of pages is least; among such splits, give the first scribe
// as little as possible, then the second, and so on. Print the split with " / ".
#include <iostream>
#include <vector>
using namespace std;

int groupsNeeded(const vector<long long>& p,
                 long long cap) {  // greedy, left to right
    int groups = 1;
    long long cur = 0;
    for (long long x : p) {
        if (cur + x > cap) groups++, cur = 0;
        cur += x;
    }
    return groups;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int cases;
    if (!(cin >> cases)) return 0;
    while (cases--) {
        int m, k;
        cin >> m >> k;
        vector<long long> p(m);
        long long lo = 0, hi = 0;  // the answer lies in (lo, hi]
        for (auto& x : p) {
            cin >> x;
            if (x > lo) lo = x;
            hi += x;
        }
        lo--;  // no share can be smaller than the thickest book
        while (hi - lo > 1) {
            long long mid = lo + (hi - lo) / 2;
            if (groupsNeeded(p, mid) <= k)
                hi = mid;
            else
                lo = mid;
        }
        long long cap = hi;
        // Fill from the right: later scribes take as much as the cap allows,
        // but leave at least one book for every scribe still to the left.
        vector<bool> slashAfter(m, false);
        int slashes = k - 1;
```

Listing 11.8 (Continued) *books1.cpp: binary search for the smallest cap, then the split rebuilt from the right.*

```cpp
        long long cur = 0;
        for (int i = m - 1; i >= 0; i--) {
            if (cur > 0 && (cur + p[i] > cap || i + 1 == slashes)) {
                slashAfter[i] = true;
                slashes--;
                cur = 0;
            }
            cur += p[i];
        }
        for (int i = 0; i < m; i++) {
            cout << p[i];
            if (i + 1 < m) cout << (slashAfter[i] ? " / " : " ");
        }
        cout << '\n';
    }
    return 0;
}
```

Listing 11.9 `books1.py`: the same search and rebuild in Python.

```
"""BOOKS1 (Copying Books): split the books in order among k scribes with the
least largest share; then give earlier scribes as little as possible."""

import sys


def groups_needed(p: list[int], cap: int) -> int:
    groups, cur = 1, 0
    for x in p:
        if cur + x > cap:
            groups, cur = groups + 1, 0
        cur += x
    return groups


def split(p: list[int], k: int) -> str:
    lo, hi = max(p) - 1, sum(p)  # the answer lies in (lo, hi]
    while hi - lo > 1:
        mid = (lo + hi) // 2
        if groups_needed(p, mid) <= k:
            hi = mid
        else:
            lo = mid
    slash_after = [False] * len(p)
    slashes, cur = k - 1, 0
    for i in range(len(p) - 1, -1, -1):  # fill from the right
        if cur > 0 and (cur + p[i] > hi or i + 1 == slashes):
            slash_after[i] = True
            slashes -= 1
            cur = 0
        cur += p[i]
    parts = []
    for i, x in enumerate(p):
        parts.append(str(x))
        if slash_after[i]:
            parts.append("/")
    return " ".join(parts)
```

Listing 11.9 (Continued) *books1.py: the same search and rebuild in Python.*

```
def main() -> None:
    data = sys.stdin.buffer.read().split()
    pos, out = 1, []
    for _ in range(int(data[0])):
        m, k = int(data[pos]), int(data[pos + 1])
        out.append(split(list(map(int, data[pos + 2 : pos + 2 + m])), k))
        pos += 2 + m
    print("\n".join(out))


main()
```

In **Listing 11.8** the condition `i + 1 == slashes` says "books $0..i$ are exactly as many as the slashes still to be placed", so the run must close now and every remaining book will be a scribe's single book. **Listing 11.9** uses the same test. We check three shelves.

```
$ printf '3\n7 3\n30 10 20 50 40 10 20\n4 4\n5 5 5 5\n6 2\n1 1 1 1 1 10\n' | ./books1
$ printf '3\n7 3\n30 10 20 50 40 10 20\n4 4\n5 5 5 5\n6 2\n1 1 1 1 1 10\n' | python3
../python/books1.py
```

```
30 10 / 20 50 / 40 10 20
5 / 5 / 5 / 5
1 1 1 1 1 / 10
30 10 / 20 50 / 40 10 20
5 / 5 / 5 / 5
1 1 1 1 1 / 10
```

On the first shelf, the smallest cap is 70: with 60 the greedy needs four runs (30 10 20, 50, 40 10, 20). The printed split gives the scribes 40, 70 and 70 pages. The first scribe cannot get less: with only 30 pages for the first scribe, the other two would have to share 150 pages with at most 70 each. The second shelf is the case where the "one book each" rule decides everything. The third shows that the thin books go together to the first scribe.

11.8 Two Pointers (HOTELS)

The problem in our words. In HOTELS (Hotels Along the Croatian coast) [8], $N \le 3 \times 10^5$ hotels stand along a coast, each with a positive price below 10^6, and a traveler has $M < 2^{31}$ euros. She wants to stay in a run of consecutive hotels, one night each, spending as much of her money as possible without exceeding M. Print the largest total not above M (the problem guarantees it is positive).

Trying all $N(N+1)/2 \approx 4.5 \times 10^{10}$ runs is out of the question. Binary search would work (for each start, search the prefix sums for the furthest end), but there is a simpler linear method.

The window. Keep a window of consecutive hotels, from index $left$ to index $right$, and its sum. Move $right$ forward one hotel at a time, adding its price. Whenever the sum exceeds M, drop hotels from the left end until it fits again. After each step, the window is the longest run ending at $right$ with sum at most M, and since all prices are positive, the longest such run has the largest sum. Record the largest sum seen.

Why may $left$ only move forward? Let $\ell(r)$ be the smallest start such that the run from $\ell(r)$ to r costs at most M. Because prices are positive, extending a run to the right makes it more expensive, so

$$\ell(r+1) \ge \ell(r) \tag{11.7}$$

and the left pointer never needs to go back. Each pointer moves at most N times, so the whole scan costs at most $2N$ steps: $O(N)$. **Figure 11.2** shows the window on five hotels with prices 2, 1, 3, 4, 5 and a budget of 12: the window grows to the sum 10, then adding the 5 makes 15, so the 2 and the 1 are dropped, leaving 3, 4, 5 with sum exactly 12.

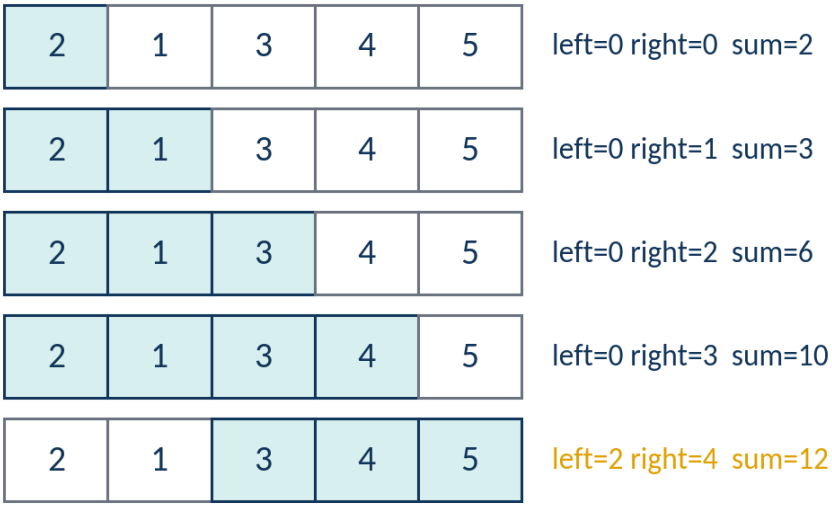


Figure 11.2 The two-pointer window on prices 2, 1, 3, 4, 5 with budget 12: the right end advances one step per row, and the left end jumps forward only when the sum exceeds the budget.

Listing 11.10 `hotels.cpp`: the two-pointer window.

```cpp
// HOTELS (Hotels Along the Croatian coast): the largest sum of consecutive
// hotel prices that is at most M. All prices are positive: two pointers.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    long long m;
    if (!(cin >> n >> m)) return 0;
    vector<long long> a(n);
    for (auto& x : a) cin >> x;
    long long sum = 0, best = 0;  // sum of the window a[left..right]
    int left = 0;
    for (int right = 0; right < n; right++) {
        sum += a[right];
        while (sum > m) sum -= a[left++];  // too expensive: drop from the left
        best = max(best, sum);
    }
    cout << best << '\n';
    return 0;
}
```

Listing 11.11 `hotels.py`: the same window in Python.

```
"""HOTELS: the largest sum of consecutive prices that is at most M.
Prices are positive, so a window moved by two pointers finds it in O(N)."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    n, m = int(data[0]), int(data[1])
    a = list(map(int, data[2 : 2 + n]))
    total = best = left = 0  # total = a[left] + ... + a[right]
    for x in a:
        total += x
        while total > m:  # too expensive: drop hotels from the left
            total -= a[left]
            left += 1
        if total > best:
            best = total
    print(best)


main()
```

Listing 11.10 keeps the sum in `long long`: a single price is below 10^6, but the running sum just after adding a price can exceed M, which may itself be close to 2^{31}, so the sum can pass the `int` limit before the inner loop brings it back. **Listing 11.11** is a direct copy of the idea. We try eight hotels with a budget of 20.

```
$ printf '8 20\n4 7 2 9 3 5 8 1\n' | ./hotels
$ printf '8 20\n4 7 2 9 3 5 8 1\n' | python3 ../python/hotels.py
```

```
19
19
```

The best run is $2 + 9 + 3 + 5 = 19$; no run of consecutive prices sums to exactly 20. Tracing the window step by step shows the two pointers at work. The window grows to $4 + 7 + 2 = 13$; adding 9 makes 22, so 4 is dropped, leaving $7 + 2 + 9 = 18$, the best so far. Adding 3 makes 21, so 7 is dropped: $2 + 9 + 3 = 14$. Adding 5 gives $2 + 9 + 3 + 5 = 19$, the new best. Adding 8 makes 27; dropping 2 and 9 leaves $3 + 5 + 8 = 16$. Adding 1 gives 17. In total the right pointer moved 8 times and the left pointer 4 times, far fewer than the 36 runs a brute force would add up.

The same pattern solves a whole family of problems whenever *extending the window can only make it worse and shrinking it can only make it better*: the shortest run with sum at least S, the longest run with at most K distinct values, the number of runs with sum at most M. It fails when that monotonicity fails, for example when prices can be negative, as Exercise 6 shows.

11.8.1 The mirror problem: the shortest run that is enough

A close cousin asks the opposite: the *shortest* run of consecutive values whose sum is *at least* S. A farmer selling mangoes at a roadside stall, with the day's sales per hour in a list, might ask for the shortest stretch of hours in which at least 50 kilograms were sold, to decide when a second helper is really needed. The window moves the same way, but the roles of growing and shrinking swap. Advance the right end and add its value; then, *while* the sum is still at least S, record the window's length as a candidate and drop the left end. Dropping stops as soon as the sum falls below S, and the smallest length recorded is the answer.

Try it on the hourly sales 4,7,2,9,3,5,8,1 with $S = 15$. The window grows to 4,7,2,9 with sum 22: record length 4, drop 4 (sum 18), record length 3, drop 7 (sum 11) and stop. Adding 3 gives 14, not enough. Adding 5 gives 19: record length 4 (2,9,3,5), drop 2 (sum 17), record 3 (9,3,5), drop 9

(sum 8). Adding 8 gives 16: record 3 (3,5,8), drop 3 (sum 13). Adding 1 gives 14. The shortest length recorded is 3. Again each pointer moves at most N times, and again the argument rests on Equation (11.7): with positive values, the best start for the next end is never to the left of the best start for this one.

11.9 Testing It: Every Search Against a Brute Force

The five programs are checked exactly like those of Chapters 9 and 10. **Table 11.1** lists the brute forces. PIE needs one extra idea because its answer is a real number: the stress driver compares numbers with a decimal point within the 10^{-3} allowed by the problem instead of character by character, and the brute force exploits a fact noted in Section 11.6, that the best piece is always one pie cut into exactly j equal parts, so it tries every v_i/j.

Table 11.1 The brute forces used to check the five programs of this chapter.

Problem	Brute force	Small inputs
AGGRCOW	every choice of C stalls	up to 8 stalls, some on a short road
EKO	every saw height from 0 to the tallest tree	up to 8 trees
PIE	every pie cut into $j = 1, \ldots, F+1$ equal parts	up to 5 pies, 8 friends
BOOKS1	every way to cut the shelf into k runs, with the tie-breaking rule	up to 8 books
HOTELS	every run of consecutive hotels	up to 10 hotels

```
$ cd tests
$ python3 stress.py 300
```

```
aggrcow   300 random inputs: C++ and Python agree with brute force
eko       300 random inputs: C++ and Python agree with brute force
pie       300 random inputs: C++ and Python agree with brute force
books1    300 random inputs: C++ and Python agree with brute force
hotels    300 random inputs: C++ and Python agree with brute force
```

The BOOKS1 line is the one that matters most: it confirms, on 300 random shelves with several test cases each, that the split rebuilt from the right obeys the tie-breaking rule exactly as the brute force applies it.

11.10 Making It Better: Speed and Library Help

Table 11.2 shows the measured CPU times on the largest inputs.

```
program          C++ (s)  Python (s)  same output
aggrcow             0.02        0.11  yes
eko                 0.04        0.37  yes
pie                 0.01        0.04  yes
books1              0.01        0.14  yes
hotels              0.02        0.06  yes
```

Table 11.2 Largest inputs and measured CPU time for the five programs.

Problem	Largest input	Work	C++ (s)	Python (s)
AGGRCOW	10^5 stalls, 1000 cows	sort + 30 checks	0.02	0.11
EKO	10^6 trees	30 checks (C++), sort + one pass (Python)	0.04	0.37
PIE	10^4 pies, 10^4 friends	100 or 60 rounds	0.01	0.04
BOOKS1	200 shelves of 500 books	33 checks each	0.01	0.14
HOTELS	3×10^5 hotels	one pass	0.02	0.06

C++ is fast everywhere. Python is comfortable for PIE, BOOKS1 and HOTELS, borderline for AGGRCOW when there are many test cases, and too slow for EKO, whose limit is a small fraction of a second [4]; even the sort-based **Listing 11.5** spends 0.37 seconds, mostly on reading and sorting a million numbers.

Let the library search for you. Many binary searches are searches in a sorted array, and both languages have them built in. In C++, `std::lower_bound` returns the first element not less than a value, `std::upper_bound` the first element greater than it, and `std::partition_point` the first element for which a monotone predicate turns false, which is exactly our boundary [9]. In Python, `bisect.bisect_left` and `bisect.bisect_right` do the same on lists, and since Python 3.10 they accept a `key` function [10]. Use them whenever the candidates are stored in a sorted list; write the loop of Section 11.2 when the candidates are a range of numbers and the check is a computation, as in all five problems here. Exercise 5 uses `lower_bound` to find the longest increasing subsequence in $O(n \log n)$, a classic that joins this chapter to Chapter 9.

11.11 Using It for Real

The signal for binary search on the answer is a sentence of the form "find the largest x such that …" or "the smallest x such that …", where checking a given x is easy and the condition is monotone. **Table 11.3** lists the shapes you will meet most often.

Table 11.3 Typical shapes of problems solved by binary search on the answer and by two pointers.

The problem asks for	The check	Direction	Example
the largest smallest gap	greedy placement from the left	largest yes	AGGRCOW
the highest level that still yields enough	a sum over all items	largest yes	EKO
the largest equal share	count the shares each item gives	largest yes, on reals	PIE
the smallest largest load	greedy packing into at most k groups	smallest yes	BOOKS1
the best consecutive run under a budget	two pointers, not binary search	linear scan	HOTELS

Everyday versions are easy to find. A bakery with three ovens and a list of trays in a fixed order wants the shortest baking day: that is BOOKS1 with ovens as scribes. A cooperative that must cut bamboo poles of different lengths into pieces of equal length for a fence, at least F pieces, asks PIE's question on a line instead of in circles. A water utility placing C monitoring stations along a pipeline so that they are as spread out as possible asks AGGRCOW's. In each case, write the check first, test it on a few values by hand to see the block of yes and the block of no, and only then add the search around it. Bentley's classic account of binary search [11] and Laaksonen's chapter on binary search in contests [12] give many more examples, including the careful handling of the ends.

11.12 Common Mistakes, and How to Avoid Them

- **An infinite loop.** `while (lo < hi)` with `lo = mid` loops forever when `hi == lo + 1` and `mid` rounds down to `lo`. The template `while (hi - lo > 1)` with mid strictly between the ends cannot loop.
- **Wrong starting values.** If lo_0 is not truly a yes, or hi_0 not truly a no, the answer can be off by one at the edges. Choose ends that are certain, even if the range becomes a little larger; it costs one extra step.

- **A check that is not monotone.** Binary search on a question whose answers go yes, no, yes finds *a* boundary, not *the* boundary. Print $ok(x)$ for a range of small x and look before you search.
- **Overflow in the middle or in the check.** Write `lo + (hi - lo) / 2`, and sum wood, pages or prices in `long long`.
- **Real-number searches that stop on a tolerance.** Use a fixed number of rounds; 100 is always enough for `double`.
- **Returning the wrong end.** For "largest yes" the answer is lo; for "smallest yes" it is hi. Decide which one before writing the loop.
- **Two pointers with negative values.** The window argument Equation (11.7) needs all values positive (or non-negative). With negative values, use prefix sums with a sorted structure instead.
- **Forgetting to sort.** AGGRCOW's stalls arrive in any order; the greedy check needs them sorted.

11.13 Chapter Summary

- When a yes/no question $ok(x)$ is monotone, binary search finds the boundary between yes and no with $\lceil \log_2(hi_0 - lo_0) \rceil$ checks (Equation (11.2)), keeping the invariant Equation (11.1) from start to finish.
- "Maximize the minimum" (AGGRCOW) and "minimize the maximum" (BOOKS1, Equation (11.6)) become binary searches around a greedy check; "the highest level that yields enough" (EKO, Equation (11.3)) is a search around a sum.
- On real numbers (PIE, Equation (11.5)) the search runs a fixed number of rounds and uses `double` and an exact π.
- Once the best value is known, the solution itself can be rebuilt; for BOOKS1 a pass from the right obeys the tie-breaking rule.
- Two pointers replace a quadratic search over runs by a linear scan when the left end never has to move back (Equation (11.7)), as in HOTELS.
- All five programs were checked against brute forces on 1500 random inputs. C++ is fast everywhere; Python is fine except for EKO's million trees.

11.14 Exercises

1. Run the binary search of **Listing 11.2** by hand for the stalls 1, 3, 7, 10, 12 and three cows. Give lo, hi and mid at every step, and count the checks.
2. EKO promises that the trees together are taller than M. What goes wrong in **Listing 11.4** if that promise is broken, and how would you detect it?
3. Write the loop for the *smallest* x with $ok(x)$ true, for a question whose answers go no, no, ..., yes, yes. Which end is returned, and what must be true of the starting values?
4. Change the two-pointer idea of HOTELS to *count* the runs of consecutive hotels whose total is at most M. Apply it by hand to the prices 2, 1, 3 with $M = 3$.
5. The longest strictly increasing subsequence of a sequence can be found in $O(n \log n)$ with `lower_bound`. Explain the idea and run **Listing 11.12** on 5,2,8,6,3,6,9,7.
6. Give an input with a negative price on which the window of **Listing 11.10** returns a wrong answer, and describe a correct $O(N \log N)$ method.

11.15 Solutions to the Exercises

1. Sorted stalls: 1, 3, 7, 10, 12, so $lo = 0$ and $hi = 12$. First $mid = 6$: cows at 1 and 7, and nothing is 6 beyond 7, so no; $hi = 6$. Then $mid = 3$: cows at 1, 7, 10, yes; $lo = 3$. Then $mid = 4$: cows at 1, 7, 12, yes; $lo = 4$. Then $mid = 5$: cows at 1, 7, 12, yes; $lo = 5$. Now $hi - lo = 1$ and the answer is 5, after four checks, $\lceil \log_2 12 \rceil = 4$ as Equation (11.2) predicts.

2. The search assumes $ok(0)$ is true. If all the trees together give less than M meters, $ok(0)$ is false, the invariant is broken from the start, and the program still prints 0, a height that does not give enough wood. To detect it, compute $wood(0)$ once before the loop and report "impossible" when it is below M. A good habit: before any binary search, check that the starting values really are a yes and a no.

3. Keep $ok(lo)$ false and $ok(hi)$ true. Loop while `hi - lo > 1`; set `mid = lo + (hi - lo) / 2`; if $ok(mid)$ then $hi = mid$ else $lo = mid$. Return hi. The starting lo must be a value that is certainly no (it may be one below the smallest possible answer, even if that value itself is impossible as an answer), and the starting hi a value that is certainly yes. **Listing 11.8** does exactly this.

4. For each right end r, after shrinking, every run that ends at r and starts at or after $left$ costs at most M, because it is part of the window and prices are positive; every run starting before $left$ costs more. So add $r - left + 1$ to the count at each step. For 2, 1, 3 with $M = 3$: at $r = 0$ the window is [2], add 1; at $r = 1$ it is [2,1] with sum 3, add 2; at $r = 2$ the sum 6 is too much, drop 2 and then 1, leaving [3], add 1. The count is 4: the runs [2], [1], [2,1] and [3].

5. Keep a list `tail` where `tail[L]` is the smallest possible last value of an increasing subsequence of length $L + 1$ seen so far. The list is always strictly increasing, so for each new value x, `lower_bound` finds the first tail not smaller than x: replacing it by x keeps a subsequence of that length with a smaller end, and if there is none, x extends the longest subsequence. The length of `tail` is the answer. Each value costs one binary search, so the total is $O(n \log n)$, against the $O(n^2)$ of the direct DP of Chapter 9.

Listing 11.12 `lis.cpp`: the longest strictly increasing subsequence with `lower_bound`.

```
// Exercise 11.5: length of the longest strictly increasing subsequence in
// O(n log n): tail[L] is the smallest possible last value of an increasing
// subsequence of length L+1, and lower_bound finds where each value goes.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> a = {5, 2, 8, 6, 3, 6, 9, 7};
    vector<int> tail;
    for (int x : a) {
        auto it = lower_bound(tail.begin(), tail.end(), x);  // first tail >= x
        if (it == tail.end())
            tail.push_back(x);
        else
            *it = x;
        cout << "after " << x << ": tails";
        for (int v : tail) cout << ' ' << v;
        cout << '\n';
    }
    cout << "longest strictly increasing subsequence: " << tail.size() << '\n';
    return 0;
}
```

```
$ ./lis
```

```
after 5: tails 5
after 2: tails 2
after 8: tails 2 8
after 6: tails 2 6
after 3: tails 2 3
after 6: tails 2 3 6
after 9: tails 2 3 6 9
after 7: tails 2 3 6 7
longest strictly increasing subsequence: 4
```

The answer is 4, for example 2,3,6,9 or 2,3,6,7. Note that `tail` is not itself a subsequence of the input in general; only its length is meaningful.

6. With prices $6, -3, 4$ and $M = 5$, the window program first sees 6, which is too much, and empties the window; then it adds -3 and 4 and ends with the window $[-3,4]$, sum 1. Its answer is 1, but the single hotel with price 4 gives 4. With a negative value, shrinking a window can make it more expensive, so Equation (11.7) fails. A correct method uses prefix sums $P_0 = 0, P_1, \ldots, P_N$: the run from $i + 1$ to r costs $P_r - P_i$, and we want the smallest $P_i \geq P_r - M$ among $i < r$. Keeping the earlier prefix sums in a balanced search tree (`std::set` in C++) and asking `lower_bound(P[r] - M)` for each r gives $O(N \log N)$.

References

[1] Sphere Online Judge, "AGGRCOW - Aggressive cows." [Online]. Available: https://www.spoj.com/problems/AGGRCOW/

[2] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998, Sec. 6.2.1.

[3] J. Bloch, "Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken," Google Research Blog, Jun. 2, 2006. [Online]. Available: https://research.google/blog/extra-extra-read-all-about-it-nearly-all-binary-searches-and-mergesorts-are-broken/

[4] Sphere Online Judge, "EKO - Eko." [Online]. Available: https://www.spoj.com/problems/EKO/

[5] Sphere Online Judge, "PIE - Pie." [Online]. Available: https://www.spoj.com/problems/PIE/

[6] *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2019, 2019.

[7] Sphere Online Judge, "BOOKS1 - Copying Books." [Online]. Available: https://www.spoj.com/problems/BOOKS1/

[8] Sphere Online Judge, "HOTELS - Hotels Along the Croatian Coast." [Online]. Available: https://www.spoj.com/problems/HOTELS/

[9] cppreference.com, "std::partition_point," accessed Sep. 2026. [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/partition_point

[10] Python Software Foundation, "bisect - Array bisection algorithm," *The Python Standard Library*, Python 3.14 documentation, accessed Sep. 2026. [Online]. Available: https://docs.python.org/3/library/bisect.html

[11] J. Bentley, *Programming Pearls*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2000, Column 4.

[12] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

Chapter 12
Graph Algorithms Toolkit

A surprising number of judge problems are graph problems in disguise: a chessboard, a map of cities, a list of chores that must happen in a certain order, a colony of bugs, a network of cables. This chapter gives you a toolkit of seven graph techniques that cover most of what you will meet on SPOJ at the beginner and intermediate level, and it builds each one around a real SPOJ problem. By the end of the chapter you will be able to store a graph from judge input in an adjacency list; decide in one minute which technique a statement is asking for; test whether a graph is a tree with union-find (PT07Y); find shortest paths with breadth-first search when every step costs the same (NAKANJ) and with Dijkstra's algorithm when steps have different costs (SHPATH); split a graph into two groups with a coloring search (BUGLIFE); measure the longest path in a tree with two searches (PT07Z); order tasks that depend on one another (TOPOSORT); and connect everything as cheaply as possible (MST). Every technique comes with a proof in plain words, its running time, a complete program in C++17 and in Python, real output on our own inputs, and a stress test against a brute force.

12.1 A Driver, a Timetable and a Chessboard

Pak Darto drives a delivery van for a laundry in Surabaya. Every morning he has a list of addresses, a map in his head of which streets connect which neighborhoods, and a rough idea of how many minutes each street takes at that hour. When a customer calls and asks, "How soon can you be here?", he is solving a shortest-path problem: from where he is now, through the streets, to the customer, with the smallest total time. He does it by feel, and after ten years he is good at it. A program can do it exactly, for ten thousand places, in a fraction of a second.

At the same time, in a secondary school across town, the deputy head is making next semester's timetable. Some subjects must come before others: the introductory physics unit before the laboratory unit, basic algebra before functions. She has a list of pairs "this before that" and needs one order that respects them all, or proof that the list contradicts itself. That is a topological sorting problem. Her colleague who plans the school's new network cable wants every classroom connected to every other at the lowest total length of cable: a minimum spanning tree.

And a twelve-year-old at the chess club asks how many moves a knight needs to get from one corner of the board to the other. That is a shortest-path problem again, but a special one, because every move costs exactly one.

All four people are working on a **graph**: a set of things (places, subjects, classrooms, squares) and a set of connections between pairs of them (streets, "before" rules, possible cables, knight moves). Competitive programming problems use the same structure, dressed in different stories. The skill this chapter builds is to see the graph behind the story, to recognize which of a small number of standard questions is being asked, and to answer it with a technique that you can write from memory without bugs. SPOJ is full of such problems, and the seven in this chapter are among the best known on the site [1].

12.2 The Idea in Plain Words

12.2.1 Vertices, edges and the words around them

> **DEFINITION.** A *graph* $G = (V, E)$ is a set V of *vertices* (also called nodes) and a set E of *edges*, each edge joining two vertices. We write $n = |V|$ for the number of vertices and $m = |E|$ for the number of edges. In an *undirected* graph an edge $\{u, v\}$ can be used both ways; in a *directed* graph an edge (u, v) goes from u to v only. In a *weighted* graph every edge carries a number, its *weight* or cost. A *path* is a sequence of vertices in which each consecutive pair is joined by an edge; a *cycle* is a path of at least one edge that returns to its start without reusing an edge. A graph is *connected* when there is a path between every pair of vertices.

Judge statements almost always number the vertices 1,2, … , n and then list the edges, one per line, as two vertex numbers (and a weight when the graph is weighted). PT07Y, for example, gives n and m on the first line and then m lines "u v". The first thing to decide is how to keep that list in memory so that the algorithm can ask the question it asks most often: *which vertices are next to this one?*

12.2.2 Storing a graph: the adjacency list

There are three common ways to store a graph, summarized in **Table 12.1**. An **edge list** keeps the m pairs exactly as they were read; it is all that Kruskal's algorithm needs, because Kruskal only sorts the edges and looks at them one by one. An **adjacency matrix** is an $n \times n$ table whose cell (u, v) says whether there is an edge (or holds its weight); it answers "is u next to v?" in one step, but it needs n^2 cells, which for $n = 10{,}000$ is 10^8 cells, 400 MB of `int`, far more than any SPOJ memory limit you will meet. An **adjacency list** keeps, for every vertex, a list of its neighbors; it uses memory proportional to $n + m$ and lets an algorithm visit all neighbors of u in time proportional to their number. For the problems of this chapter the adjacency list is the default choice, and in C++ it is simply a `vector<vector<int>>` (or `vector<vector<pair<int,int>>>` for weights), in Python a list of lists.

Table 12.1 Three ways to store a graph, with the cost of the two questions algorithms ask most.

Representation	Memory	"Neighbors of u"	"Is (u,v) an edge?"	Used here by
Edge list	$O(m)$	$O(m)$	$O(m)$	Kruskal (MST)
Adjacency matrix	$O(n^2)$	$O(n)$	$O(1)$	small n only (brute forces)
Adjacency list	$O(n + m)$	$O(\deg u)$	$O(\deg u)$	BFS, Dijkstra, Kahn, two-coloring

Two details save many Wrong Answers. First, in an undirected graph every edge must be added *twice*, once to the list of each end; forgetting the second direction is one of the classic mistakes of Chapter 7. Second, allocate the lists with $n + 1$ entries and use indices $1..n$ directly, so that you never have to remember to subtract one; the wasted entry 0 costs nothing.

12.2.3 Seven questions, seven tools

Almost every graph problem at this level asks one of a handful of questions. **Figure 12.1** is the map of this chapter: find the question on the left, and the tool on the right is what the rest of the chapter builds, with the SPOJ problem we use to practice it.

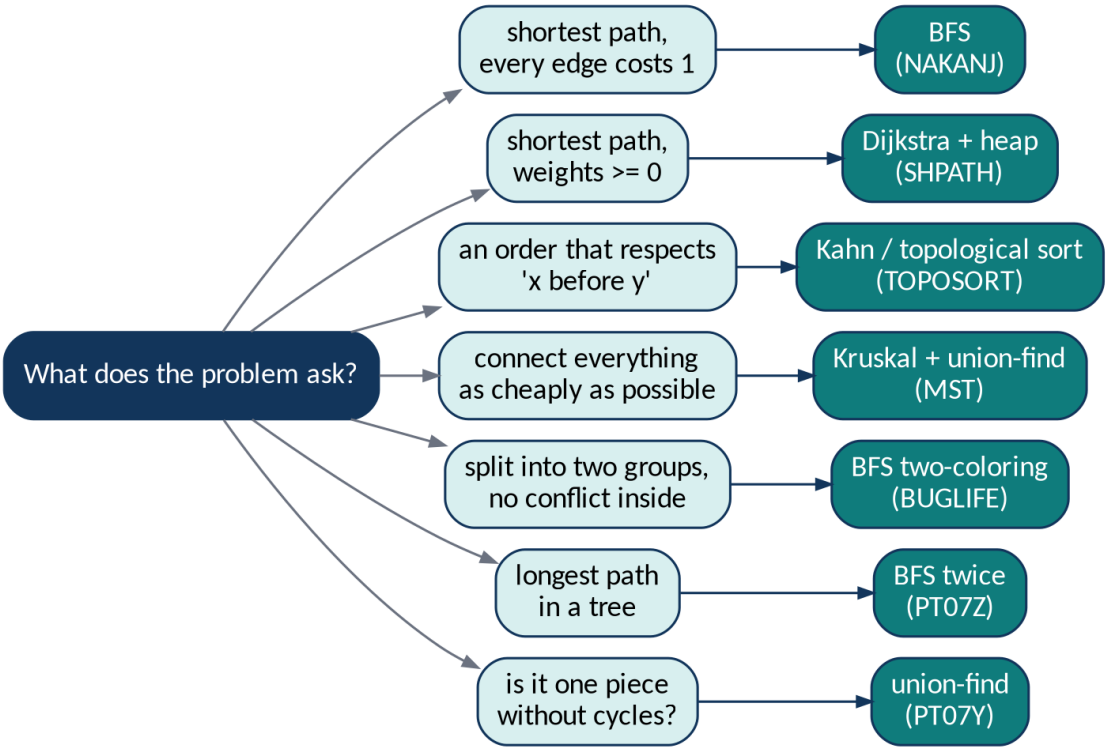


Figure 12.1 The graph toolkit of this chapter: the question a statement asks, the technique that answers it, and the SPOJ problem used to practice it.

The order of the sections follows the difficulty of the proofs rather than the order in the figure. We start with union-find, which needs no search at all, move on to breadth-first search and two of its cousins, then Dijkstra, topological sorting and minimum spanning trees. Keep the complexity budget of Chapter 3 in mind: with n up to 10^4 and m up to 10^6, every algorithm here runs in time close to $O(n+m)$ or $O((n+m)\log n)$, which is a few million to a few tens of millions of basic steps, well inside a second in C++.

12.3 Is It a Tree? Union-Find (PT07Y)

12.3.1 The problem in our own words

SPOJ problem PT07Y, "Is it a tree" [2], gives an undirected graph with $n \le 10{,}000$ vertices and $m \le 20{,}000$ edges and asks for one word: `YES` if the graph is a tree, `NO` otherwise.

> **DEFINITION.** A *tree* is a connected undirected graph with no cycle. Equivalently, it is a graph in which there is exactly one path between every pair of vertices.

12.3.2 The idea

A classical fact about trees gives a short test. Every tree on n vertices has exactly $n-1$ edges, and conversely, a graph with n vertices, exactly $n-1$ edges and no cycle is connected, hence a tree. Why? Start with n separate vertices, which form n separate pieces (the word for a piece is *component*). Adding an edge that does not close a cycle always joins two different pieces into one, so it reduces the number of pieces by exactly one. After $n-1$ such edges, $n-(n-1)=1$ piece remains: the graph is connected.

So the whole test is: the number of edges must be $n-1$, and no edge may close a cycle. An edge $\{u,v\}$ closes a cycle exactly when u and v are already in the same piece. We need a structure that

keeps track of pieces while edges arrive and answers "are u and v in the same piece?" quickly. That structure is **union-find**, also called a disjoint-set union (DSU) [3], [4].

12.3.3 How union-find works

Every piece is stored as a small tree of parent pointers, and the root of that tree is the piece's representative. `find(x)` follows parent pointers from x up to the root. `unite(a, b)` finds both roots; if they are equal, a and b were already connected and the function reports it; otherwise it hangs one root below the other. Two tricks keep the trees flat. *Union by size* hangs the smaller tree below the larger, so a vertex's depth can grow only when its piece at least doubles, which caps the depth at $\log_2 n$. *Path halving* makes every vertex on the way to the root point to its grandparent, so that the next `find` is shorter. With both tricks, a sequence of m operations on n elements costs $O(m\,\alpha(n))$, where α is the inverse Ackermann function, a function so slowly growing that $\alpha(n) \le 4$ for any n you could ever store [3], [5]. In practice: constant time per operation.

Listing 12.1 is the complete program. The `DSU` structure is fourteen lines that you will reuse in Section 12.9 and in many problems after this book.

Listing 12.1 `pt07y.cpp`: a graph is a tree when it has $n - 1$ edges and none of them closes a cycle.

```cpp
// PT07Y - Is it a tree? Union-find: N-1 edges and no edge closes a cycle.
#include <cstdio>
#include <numeric>
#include <utility>
#include <vector>
using namespace std;

struct DSU {
    vector<int> parent, size;
    explicit DSU(int n) : parent(n + 1), size(n + 1, 1) {
        iota(parent.begin(), parent.end(), 0);
    }
    int find(int x) {
        while (parent[x] != x) {
            parent[x] = parent[parent[x]];  // path halving
            x = parent[x];
        }
        return x;
    }
    bool unite(int a, int b) {  // false when a and b were already connected
        a = find(a);
        b = find(b);
        if (a == b) return false;
        if (size[a] < size[b]) swap(a, b);
        parent[b] = a;
        size[a] += size[b];
        return true;
    }
};

int main() {
    int n, m;
    if (scanf("%d %d", &n, &m) != 2) return 0;
    DSU dsu(n);
    bool ok = (m == n - 1);
    for (int i = 0; i < m; i++) {
        int u, v;
        if (scanf("%d %d", &u, &v) != 2) return 0;
        if (!dsu.unite(u, v)) ok = false;  // a cycle, or a self-loop
    }
    puts(ok ? "YES" : "NO");
    return 0;
}
```

Notice that the program keeps reading all m edges even after it has seen a cycle. On a problem with a single test case it could stop early, but on a problem with several test cases, stopping in the middle of one test would make the program read the rest of that test as the start of the next one; reading every line of input is a habit worth keeping. A self-loop "$u\ u$" is handled for free: `unite(u, u)` finds the same root twice and reports a cycle.

Listing 12.2 is the same idea in Python, with the union-find written inline as a list `parent` and a nested function `find`. Python reads the whole input at once with `sys.stdin.buffer.read().split()`, the fast-input habit of Chapter 4.

Listing 12.2 `pt07y.py`: the same test in Python.

```
# PT07Y - Is it a tree? Union-find: N-1 edges and no edge closes a cycle.
import sys


def main():
    data = sys.stdin.buffer.read().split()
    n, m = int(data[0]), int(data[1])
    parent = list(range(n + 1))

    def find(x):
        while parent[x] != x:
            parent[x] = parent[parent[x]]  # path halving
            x = parent[x]
        return x

    ok = m == n - 1
    for i in range(m):
        a = find(int(data[2 + 2 * i]))
        b = find(int(data[3 + 2 * i]))
        if a == b:
            ok = False  # a cycle, or a self-loop
        else:
            parent[a] = b
    print("YES" if ok else "NO")


main()
```

12.3.4 Running it

Our own first example is a small tree: vertex 2 in the middle, joined to 1, 3 and 4.

```
4 3
1 2
2 3
2 4
```

```
YES
```

The second example has three edges on four vertices, the right count for a tree, but the edges form a triangle $1 - 2 - 3$ and vertex 4 hangs alone. The edge count alone would be fooled; the cycle check is not.

```
4 3
1 2
2 3
3 1
```

```
NO
```

12.4 Breadth-First Search: Knight Moves (NAKANJ)

12.4.1 The problem in our own words

SPOJ problem NAKANJ, “Minimum Knight moves !!!” [6], gives up to 4,096 pairs of squares on an ordinary 8×8 chessboard, written in chess notation such as `a1` and `h8`, and asks for the smallest number of knight moves from the first square to the second. A knight moves two squares in one direction and one square at a right angle, so from a square in the middle of the board it has eight possible moves.

12.4.2 The idea: explore in rings

Think of the board as a graph with 64 vertices, one per square, and an edge between two squares whenever a knight can jump from one to the other. Every edge costs one move, so the question is the length of a shortest path in an *unweighted* graph. **Breadth-first search** (BFS) answers exactly that question [3]. It explores the graph in rings: first the start square (distance 0), then every square one move away (distance 1), then every square one move away from those that has not been seen yet (distance 2), and so on. A first-in, first-out queue keeps the rings in order: squares enter the queue in order of distance, so they leave it in order of distance.

> **KEY INSIGHT.** When BFS assigns a distance to a vertex, that distance is final. The vertex is reached for the first time from a vertex of the previous ring, and no shorter route can exist, because every vertex of every earlier ring has already been taken out of the queue and examined. In symbols, if $d(v)$ is the distance BFS assigns, then for every edge (u, v) we have $d(v) \le d(u) + 1$, and the first assignment achieves the smallest possible value.

Each vertex enters the queue at most once and each edge is looked at at most twice, so BFS runs in $O(n + m)$ time. On the chessboard that is at most 64 squares and 64×8 edges per query, a few hundred steps; 4,096 queries cost about two million steps, nothing at all.

Listing 12.3 converts the letter `a`–`h` to a column $0..7$ and the digit `1`–`8` to a row $0..7$, then runs one BFS per query. The arrays `DR` and `DC` hold the eight knight moves; the distance table doubles as the "seen" marker, with -1 meaning "not reached yet". The search stops as soon as the target leaves the queue.

Listing 12.3 `nakanj.cpp`: one breadth-first search per query on the 64 squares of the board.

```cpp
// NAKANJ - Minimum knight moves: breadth-first search on the 8 x 8 board.
#include <cstdio>
#include <queue>
#include <utility>
using namespace std;

const int DR[8] = {1, 2, 2, 1, -1, -2, -2, -1};
const int DC[8] = {2, 1, -1, -2, -2, -1, 1, 2};

int knightDistance(int r0, int c0, int r1, int c1) {
    int dist[8][8];
    for (auto& row : dist)
        for (int& d : row) d = -1;  // -1 means "not reached yet"
    queue<pair<int, int>> q;
    dist[r0][c0] = 0;
    q.push({r0, c0});
    while (!q.empty()) {
        auto [r, c] = q.front();
        q.pop();
        if (r == r1 && c == c1) return dist[r][c];
        for (int k = 0; k < 8; k++) {
            int nr = r + DR[k], nc = c + DC[k];
            if (nr < 0 || nr >= 8 || nc < 0 || nc >= 8) continue;
            if (dist[nr][nc] != -1) continue;
            dist[nr][nc] = dist[r][c] + 1;
            q.push({nr, nc});
        }
    }
    return -1;  // never happens: a knight reaches every square
}

int main() {
    int t;
    if (scanf("%d", &t) != 1) return 0;
    while (t--) {
        char a[3], b[3];
        if (scanf("%2s %2s", a, b) != 2) break;
        int d = knightDistance(a[1] - '1', a[0] - 'a', b[1] - '1', b[0] - 'a');
        printf("%d\n", d);
    }
    return 0;
}
```

Listing 12.4 `nakanj.py`: the same search in Python, with `collections.deque` as the queue.

```python
# NAKANJ - Minimum knight moves: breadth-first search on the 8 x 8 board.
import sys
from collections import deque

MOVES = [(1, 2), (2, 1), (2, -1), (1, -2), (-1, -2), (-2, -1), (-2, 1), (-1, 2)]


def knight_distance(r0, c0, r1, c1):
    dist = [[-1] * 8 for _ in range(8)]  # -1 means "not reached yet"
    dist[r0][c0] = 0
    q = deque([(r0, c0)])
    while q:
        r, c = q.popleft()
        if (r, c) == (r1, c1):
            return dist[r][c]
        for dr, dc in MOVES:
            nr, nc = r + dr, c + dc
            if 0 <= nr < 8 and 0 <= nc < 8 and dist[nr][nc] == -1:
                dist[nr][nc] = dist[r][c] + 1
                q.append((nr, nc))
    return -1  # never happens: a knight reaches every square


def main():
    data = sys.stdin.read().split()
    t = int(data[0])
    out = []
    for i in range(t):
        a, b = data[1 + 2 * i], data[2 + 2 * i]
        out.append(knight_distance(ord(a[1]) - 49, ord(a[0]) - 97,
                                   ord(b[1]) - 49, ord(b[0]) - 97))
    print("\n".join(map(str, out)))


main()
```

In Python (**Listing 12.4**), use `collections.deque` and never `list.pop(0)`: removing the first element of a Python list shifts all the others and turns an $O(n+m)$ search into an $O(n^2)$ one. The C++ program uses `std::queue`, which is a deque underneath.

12.4.3 Running it

```
3
a1 h8
b1 c3
d4 d4
```

```
6
1
0
```

Corner to opposite corner takes six moves; `b1` to `c3` is a single jump; and a square to itself takes zero moves, an edge case that a program which only reports distances *after* a move would get wrong. Since there are only 64 squares, an alternative is to run 64 searches once at the start and then answer every query from a 64×64 table; that is the kind of precomputation Chapter 3 recommends whenever the number of distinct questions is small.

12.5 Two Groups Without Conflict: Bipartite Graphs (BUGLIFE)

12.5.1 The problem in our own words

SPOJ problem BUGLIFE, "A Bug's Life" [7], describes a biologist who watches pairs of bugs interacting and assumes that every interaction is between two bugs of different sexes. Each scenario gives up to 2,000 bugs and up to 1,000,000 interactions; the program must print `Scenario #i:` followed by `Suspicious bugs found!` if the observations contradict the assumption, and `No suspicious bugs found!` otherwise. Strip away the story and the question is: can the vertices of the graph be colored with two colors so that every edge joins two different colors?

> **DEFINITION.** A graph is *bipartite* (two-colorable) when its vertices can be split into two groups so that every edge joins a vertex of one group to a vertex of the other.

12.5.2 The idea

Color the start vertex 0. Its neighbors are then forced to be color 1, their neighbors color 0, and so on: in a connected graph, the color of the first vertex decides all the others. BFS is a convenient way to spread the colors, ring by ring. If the search ever finds an edge whose two ends already have the same color, the graph is not bipartite; if it finishes without such an edge, the coloring it built is a valid one. A graph is bipartite exactly when it has no cycle of odd length [5]: going around an odd cycle, the colors would have to alternate an odd number of times and come back different from where they started.

The graph need not be connected. Some bugs may never interact with others, and the interactions may form several separate groups. The outer loop in **Listing 12.5** therefore starts a new search from every vertex that is still uncolored; forgetting that loop, and searching only from vertex 1, is the most common Wrong Answer on this problem.

Listing 12.5 `buglife.cpp`: BFS two-coloring on every connected component, with a byte-buffer reader for the million-line input.

```cpp
// BUGLIFE - A bug's life: is every interaction graph two-colorable (bipartite)?
#include <cstdio>
#include <queue>
#include <vector>
using namespace std;

static char buf[1 << 16];
static size_t bufLen = 0, bufPos = 0;

int readChar() {
    if (bufPos == bufLen) {
        bufLen = fread(buf, 1, sizeof(buf), stdin);
        bufPos = 0;
        if (bufLen == 0) return -1;
    }
    return buf[bufPos++];
}

bool readInt(int& x) {
    int c = readChar();
    while (c != -1 && (c < '0' || c > '9')) c = readChar();
    if (c == -1) return false;
    x = 0;
    for (; c >= '0' && c <= '9'; c = readChar()) x = x * 10 + (c - '0');
    return true;
}

bool bipartite(const vector<vector<int>>& adj) {
    int n = (int)adj.size() - 1;
    vector<int> color(n + 1, -1);  // -1 = not colored yet, otherwise 0 or 1
    queue<int> q;
    for (int s = 1; s <= n; s++) {
        if (color[s] != -1) continue;  // a new connected component starts at s
        color[s] = 0;
        q.push(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int v : adj[u]) {
                if (color[v] == -1) {
                    color[v] = 1 - color[u];
                    q.push(v);
                } else if (color[v] == color[u]) {
                    return false;  // an edge inside one color class
                }
            }
```

Listing 12.5 (Continued) *buglife.cpp: BFS two-coloring on every connected component, with a byte-buffer reader for the million-line input.*

```
        }
    }
    return true;
}

int main() {
    int scenarios;
    if (!readInt(scenarios)) return 0;
    for (int sc = 1; sc <= scenarios; sc++) {
        int n, m;
        if (!readInt(n) || !readInt(m)) return 0;
        vector<vector<int>> adj(n + 1);
        for (int i = 0; i < m; i++) {
            int a, b;
            if (!readInt(a) || !readInt(b)) return 0;
            adj[a].push_back(b);
            adj[b].push_back(a);
        }
        printf("Scenario #%d:\n", sc);
        bool ok = bipartite(adj);
        puts(ok ? "No suspicious bugs found!" : "Suspicious bugs found!");
    }
    return 0;
}
```

With up to a million interaction lines per scenario, input speed matters, and the program uses the `fread` buffer reader of Chapter 4 instead of `scanf`. The work per scenario is $O(n + m)$. **Listing 12.6** does the same in Python.

Listing 12.6 `buglife.py`: the same two-coloring in Python.

```python
# BUGLIFE - A bug's life: is every interaction graph two-colorable (bipartite)?
import sys
from collections import deque


def bipartite(n, adj):
    color = [-1] * (n + 1)  # -1 = not colored yet, otherwise 0 or 1
    for s in range(1, n + 1):
        if color[s] != -1:
            continue  # a new connected component starts at s
        color[s] = 0
        q = deque([s])
        while q:
            u = q.popleft()
            for v in adj[u]:
                if color[v] == -1:
                    color[v] = 1 - color[u]
                    q.append(v)
                elif color[v] == color[u]:
                    return False  # an edge inside one color class
    return True


def main():
    data = sys.stdin.buffer.read().split()
    pos = 1
    out = []
    for sc in range(1, int(data[0]) + 1):
        n, m = int(data[pos]), int(data[pos + 1])
        pos += 2
        adj = [[] for _ in range(n + 1)]
        for _ in range(m):
            a, b = int(data[pos]), int(data[pos + 1])
            pos += 2
            adj[a].append(b)
            adj[b].append(a)
        out.append("Scenario #%d:" % sc)
        out.append("No suspicious bugs found!" if bipartite(n, adj)
                   else "Suspicious bugs found!")
    print("\n".join(out))


main()
```

12.5.3 Running it

Our example has two scenarios: four bugs whose interactions form a cycle of length 4 (even, so two-colorable) and three bugs forming a triangle (odd, so not).

```
2
4 4
1 2
2 3
3 4
4 1
3 3
1 2
2 3
3 1
```

```
Scenario #1:
No suspicious bugs found!
Scenario #2:
Suspicious bugs found!
```

12.6 The Longest Path in a Tree (PT07Z)

12.6.1 The problem in our own words

SPOJ problem PT07Z, "Longest path in a tree" [8], gives a tree with $n \le 10{,}000$ vertices as $n - 1$ edges and asks for the number of edges on the longest path between any two vertices. That number is called the *diameter* of the tree.

12.6.2 The idea: two searches

A brute force would run BFS from every vertex and take the largest distance ever found: n searches of $O(n)$ each, $O(n^2)$ in total, 10^8 steps for $n = 10^4$. That might pass in C++, but there is a far better way that needs only two searches:

1. Run BFS from any vertex s and let a be a vertex farthest from s.
2. Run BFS from a and let b be a vertex farthest from a. The distance from a to b is the diameter.

Why does the first search always end at one end of a longest path? The method is well known [5], and the heart of the proof is a gluing argument. Take a longest path $x - \cdots - y$ of length D and suppose the farthest vertex a from s were *not* the end of any longest path. Follow the path from s to a. Wherever it leaves the longest path (or, if it never touches it, wherever it leaves the way from s to that path), the remaining part towards a is at least as long as the remaining part towards y, because a is at least as far from s as y is. Gluing the piece from x to the leaving point onto the piece from the leaving point to a therefore gives a path of length at least D that ends at a, and in the second case even longer than D: a contradiction either way. So some longest path ends at a, and the second search, which finds the farthest vertex from a, finds its other end. Exercise 12.4 asks you to write the two cases out in full, and its solution in Section 12.15 does so.

In a BFS, vertices leave the queue in order of distance, so the *last* vertex to leave the queue is a farthest one. **Listing 12.7** uses that observation instead of scanning the distance array a second time, and **Listing 12.8** does the same in Python.

Listing 12.7 `pt07z.cpp`: the diameter of a tree from two breadth-first searches.

```cpp
// PT07Z - Longest path in a tree: two breadth-first searches find the diameter.
#include <cstdio>
#include <queue>
#include <utility>
#include <vector>
using namespace std;

// Returns the vertex farthest from s and its distance (in edges).
pair<int, int> farthest(const vector<vector<int>>& adj, int s) {
    vector<int> dist(adj.size(), -1);
    queue<int> q;
    dist[s] = 0;
    q.push(s);
    int last = s;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        last = u;  // BFS pops vertices in order of distance: the last is farthest
        for (int v : adj[u])
            if (dist[v] == -1) {
                dist[v] = dist[u] + 1;
                q.push(v);
            }
    }
    return {last, dist[last]};
}

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    vector<vector<int>> adj(n + 1);
    for (int i = 0; i < n - 1; i++) {
        int u, v;
        if (scanf("%d %d", &u, &v) != 2) return 0;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    int a = farthest(adj, 1).first;  // one end of a longest path
    printf("%d\n", farthest(adj, a).second);
    return 0;
}
```

Listing 12.8 `pt07z.py`: the same two searches in Python.

```python
# PT07Z - Longest path in a tree: two breadth-first searches find the diameter.
import sys
from collections import deque


def farthest(adj, s):
    """Return the vertex farthest from s and its distance (in edges)."""
    dist = [-1] * len(adj)
    dist[s] = 0
    q = deque([s])
    last = s
    while q:
        u = q.popleft()
        last = u  # BFS pops vertices in order of distance: the last is farthest
        for v in adj[u]:
            if dist[v] == -1:
                dist[v] = dist[u] + 1
                q.append(v)
    return last, dist[last]


def main():
    data = sys.stdin.buffer.read().split()
    n = int(data[0])
    adj = [[] for _ in range(n + 1)]
    for i in range(n - 1):
        u, v = int(data[1 + 2 * i]), int(data[2 + 2 * i])
        adj[u].append(v)
        adj[v].append(u)
    a, _ = farthest(adj, 1)  # one end of a longest path
    print(farthest(adj, a)[1])


main()
```

12.6.3 Running it

The tree in our example has a central vertex 2 with branches to 1, 3 and a longer branch $4 - 5 - 6$. The longest path, $1 - 2 - 4 - 5 - 6$ (or $3 - 2 - 4 - 5 - 6$), has 4 edges.

```
6
1 2
2 3
2 4
4 5
5 6
```

```
4
```

A depth-first search written with recursion would also work, but a tree with 10,000 vertices can be a single long chain, and a recursion 10,000 levels deep is close to the default stack limit on some systems and far beyond Python's default recursion limit of 1,000. BFS with an explicit queue has no such risk, which is the reason this chapter prefers it wherever it suffices.

12.7 Shortest Paths with Weights: Dijkstra (SHPATH)

12.7.1 The problem in our own words

SPOJ problem SHPATH, "The Shortest Path" [9], describes a country with up to 10,000 cities. For every city the input gives its name (up to ten lowercase letters), the number of its neighbors, and for each neighbor its index and the positive integer cost of travel. Then come up to 100 queries,

each a pair of city names, and for each query the program prints the minimum total cost. There are up to 10 such test cases in one input, and the statement promises that no minimum cost exceeds 200,000.

This is Pak Darto's problem from Section 12.1: shortest paths, but now the edges have different costs, so BFS is no longer correct. A route with three cheap streets can beat a route with one expensive street.

12.7.2 The idea: settle the nearest city first

Dijkstra's algorithm [10] keeps, for every vertex v, a tentative distance $d(v)$: the cost of the best route to v found so far. At the start $d(s) = 0$ for the source s and $d(v) = \infty$ for everything else. The algorithm repeats one step: take the unfinished vertex u with the smallest tentative distance, declare it finished, and try to improve each neighbor v through u:

$$d(v) \leftarrow \min(d(v),\ d(u) + w(u, v)) \tag{12.1}$$

where $w(u, v)$ is the cost of the edge. This update is called *relaxing* the edge (u, v). Equation (12.1) is the same inequality that BFS maintains, with the edge weight in place of 1.

> **KEY INSIGHT.** When all weights are non-negative, the unfinished vertex with the smallest tentative distance cannot be improved later. Any other route to it would have to leave the finished region through some unfinished vertex x with $d(x) \geq d(u)$, and then add edges of non-negative cost, so it costs at least $d(u)$. This is exactly where a negative edge would break the algorithm, and why Dijkstra's algorithm must never be used when some weight is negative.

Figure 12.2 shows the finished distances on the small map we use as a test below. From Surabaya, Malang is reached more cheaply through Sidoarjo ($1 + 2 = 3$) than directly (4), and Jember through Sidoarjo and Malang ($1 + 2 + 3 = 6$) rather than through Sidoarjo alone ($1 + 7 = 8$).

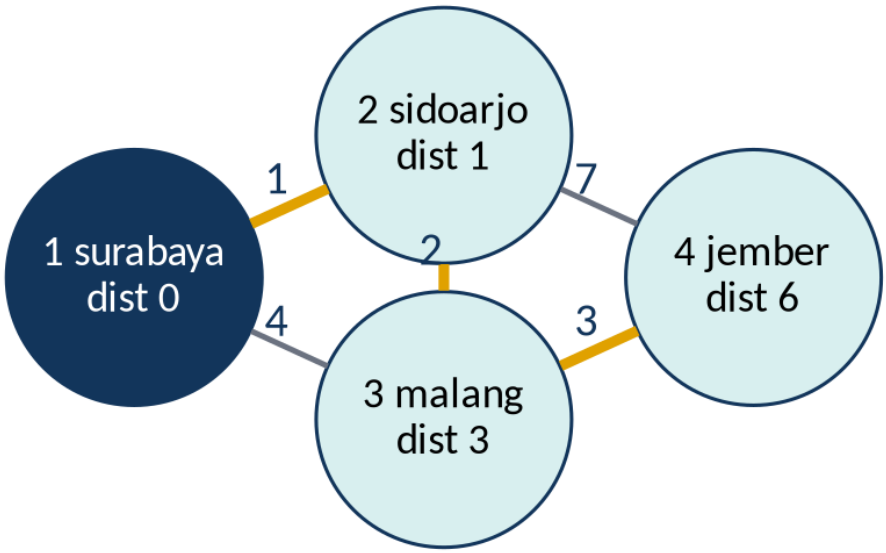


Figure 12.2 Our four-city test map with the resulting Dijkstra distances from Surabaya; the highlighted edges form the shortest-path tree.

12.7.3 A heap with lazy deletion

"Take the unfinished vertex with the smallest distance" is a job for a priority queue. The C++ `std::priority_queue` and Python's `heapq` cannot lower the key of an element already inside, so

both programs use the standard trick called *lazy deletion*: whenever a distance improves, push a new pair $(d(v), v)$; when a pair is popped whose distance no longer equals the current $d(v)$, it is stale, and the program simply skips it. The heap can hold up to m pairs, so each push and pop costs $O(\log m) = O(\log n)$, and one run of Dijkstra costs

$$T_{\text{Dijkstra}} = O((n + m)\log n). \tag{12.2}$$

Two more details matter for SHPATH. The target is final the moment it is popped, so the search can stop there instead of finishing the whole graph. And the city names must be turned into indices; an `unordered_map<string,int>` (a Python `dict`) does it in constant expected time per name.

Listing 12.9 `shpath.cpp`: Dijkstra with a binary heap and lazy deletion, one search per query, stopping at the target.

```
// SHPATH - The shortest path: Dijkstra with a binary heap, one run per query.
#include <cstdio>
#include <functional>
#include <queue>
#include <string>
#include <unordered_map>
#include <utility>
#include <vector>
using namespace std;

typedef pair<int, int> PII;  // heap: (distance, vertex); adj: (vertex, cost)

int dijkstra(const vector<vector<PII>>& adj, int s, int t) {
    const int INF = 1 << 30;
    vector<int> dist(adj.size(), INF);
    priority_queue<PII, vector<PII>, greater<PII>> pq;  // smallest distance on top
    dist[s] = 0;
    pq.push({0, s});
    while (!pq.empty()) {
        auto [d, u] = pq.top();
        pq.pop();
        if (d != dist[u]) continue;  // stale entry: u was improved after the push
        if (u == t) return d;        // the target is final once it is popped
        for (auto [v, w] : adj[u]) {
            if (d + w < dist[v]) {
                dist[v] = d + w;
                pq.push({dist[v], v});
            }
        }
    }
    return dist[t];
}

int main() {
    int s;
    if (scanf("%d", &s) != 1) return 0;
    while (s--) {
        int n;
        if (scanf("%d", &n) != 1) return 0;
        vector<vector<PII>> adj(n + 1);
        unordered_map<string, int> id;
        char name[16];
        for (int i = 1; i <= n; i++) {
            int p;
            if (scanf("%15s %d", name, &p) != 2) return 0;
            id[name] = i;
```

Listing 12.9 (Continued) *shpath.cpp: Dijkstra with a binary heap and lazy deletion, one search per query, stopping at the target.*

```
            while (p--) {
                int nr, cost;
                if (scanf("%d %d", &nr, &cost) != 2) return 0;
                adj[i].push_back({nr, cost});
            }
        }
        int r;
        if (scanf("%d", &r) != 1) return 0;
        while (r--) {
            char a[16], b[16];
            if (scanf("%15s %15s", a, b) != 2) return 0;
            printf("%d\n", dijkstra(adj, id[a], id[b]));
        }
    }
    return 0;
}
```

Listing 12.10 `shpath.py`: the same algorithm with `heapq`.

```python
# SHPATH - The shortest path: Dijkstra with a binary heap, one run per query.
import sys
from heapq import heappop, heappush


def dijkstra(adj, s, t):
    dist = [1 << 30] * len(adj)
    dist[s] = 0
    pq = [(0, s)]
    while pq:
        d, u = heappop(pq)
        if d != dist[u]:
            continue  # stale entry: u was improved after the push
        if u == t:
            return d  # the target is final once it is popped
        for v, w in adj[u]:
            if d + w < dist[v]:
                dist[v] = d + w
                heappush(pq, (d + w, v))
    return dist[t]


def main():
    data = sys.stdin.buffer.read().split()
    pos = 0
    s = int(data[pos]); pos += 1
    out = []
    for _ in range(s):
        n = int(data[pos]); pos += 1
        adj = [[] for _ in range(n + 1)]
        ident = {}
        for i in range(1, n + 1):
            ident[data[pos]] = i
            p = int(data[pos + 1]); pos += 2
            for _ in range(p):
                adj[i].append((int(data[pos]), int(data[pos + 1])))
                pos += 2
        r = int(data[pos]); pos += 1
        for _ in range(r):
            a, b = data[pos], data[pos + 1]; pos += 2
            out.append(dijkstra(adj, ident[a], ident[b]))
    print("\n".join(map(str, out)))


main()
```

Listing 12.10 is a line-by-line translation with `heapq`, which is always a min-heap. In C++, `greater<PII>` turns the default max-heap into a min-heap; forgetting it gives a program that always expands the *farthest* vertex first, which still terminates and still prints numbers, but wrong ones on most inputs. The distances fit in an `int`, because the statement bounds every answer by 200,000; in a problem without such a promise, use `long long`.

12.7.4 Running it, and the cost of 1,000 queries

The input for **Figure 12.2** lists each city with its neighbors; note that an undirected road appears twice, once in the list of each end, exactly as the SHPATH format does.

```
1
4
surabaya
2
2 1
3 4
sidoarjo
3
1 1
3 2
4 7
malang
3
1 4
2 2
4 3
jember
2
2 7
3 3
3
surabaya jember
sidoarjo malang
jember surabaya
```

```
6
2
6
```

Now the budget. With the statement's limits there can be $10 \times 100 = 1{,}000$ queries, each a Dijkstra run of cost Equation (12.2) on up to $n = 10^4$ vertices. Our maximum-size generator (Section 12.10) builds 10 tests with 10,000 cities and about 30,000 roads each, and on it the C++ program needs about 5 seconds and the Python program about 27. How dense the judge's real data are, the statement does not say. Two improvements make the program safe either way. First, many queries often share a source: compute the whole distance array once per distinct source (without stopping at the target) and answer all queries with that source from it. Second, when the graph is sparse and the queries are many, the early stop at the target already cuts the average work substantially. Exercise 12.6 asks you to add the first improvement.

12.8 Doing Things in Order: Topological Sorting (TOPOSORT)

12.8.1 The problem in our own words

SPOJ problem TOPOSORT, "Topological Sorting" [11], gives $n \le 10{,}000$ jobs and $m \le 1{,}000{,}000$ rules "x must be done before y". The program must print an order of all jobs that respects every rule; when several orders exist, it must print the *lexicographically smallest*, the one that is smallest when compared number by number from the left. When no order exists, because the rules contain a cycle such as "1 before 2, 2 before 1", it must print `Sandro fails.`. The time limit is tight, half a second.

> **DEFINITION.** A *topological order* of a directed graph is an arrangement of its vertices in a row such that every edge (x, y) points from left to right. A directed graph has a topological order if and only if it has no directed cycle; such a graph is called a DAG (directed acyclic graph).

12.8.2 Kahn's algorithm

The *in-degree* of a vertex is the number of edges pointing into it: the number of jobs that must come before it. A job with in-degree 0 is *ready*: nothing stops us from doing it now. **Kahn's**

algorithm [12] repeats: take a ready job, write it down, and remove its outgoing edges, which lowers the in-degree of each of its successors by one; any successor whose in-degree reaches 0 becomes ready. If every job is eventually written down, the list is a topological order, because a job is written only after all its predecessors. If the ready set becomes empty while jobs remain, every remaining job still waits for another remaining job, and following those "waits for" links backwards must eventually repeat a job: there is a cycle.

To get the lexicographically smallest order, keep the ready jobs in a min-heap and always take the smallest. This greedy choice is correct by an exchange argument in the style of Chapter 10: the first number of the answer must be some ready job, and choosing the smallest ready job can never hurt, because any valid order that starts with a larger job can be compared at that position and is larger. The same argument repeats at every step. With the heap, the algorithm costs $O(n\log n + m)$.

Listing 12.11 uses the `fread` reader, because a million edge lines under a 0.5-second limit is exactly the situation Chapter 4 prepares you for.

Listing 12.11 `toposort.cpp`: Kahn's algorithm with a min-heap of ready jobs, and cycle detection by counting.

```cpp
// TOPOSORT - Lexicographically smallest topological order: Kahn with a min-heap.
#include <cstdio>
#include <functional>
#include <queue>
#include <vector>
using namespace std;

static char buf[1 << 16];
static size_t bufLen = 0, bufPos = 0;

int readChar() {
    if (bufPos == bufLen) {
        bufLen = fread(buf, 1, sizeof(buf), stdin);
        bufPos = 0;
        if (bufLen == 0) return -1;
    }
    return buf[bufPos++];
}

bool readInt(int& x) {  // skips anything that is not a digit or a minus sign
    int c = readChar();
    while (c != -1 && c != '-' && (c < '0' || c > '9')) c = readChar();
    if (c == -1) return false;
    bool neg = (c == '-');
    if (neg) c = readChar();
    x = 0;
    for (; c >= '0' && c <= '9'; c = readChar()) x = x * 10 + (c - '0');
    if (neg) x = -x;
    return true;
}

int main() {
    int n, m;
    if (!readInt(n) || !readInt(m)) return 0;
    vector<vector<int>> adj(n + 1);
    vector<int> indeg(n + 1, 0);
    for (int i = 0; i < m; i++) {
        int x, y;
        if (!readInt(x) || !readInt(y)) return 0;
        adj[x].push_back(y);
        indeg[y]++;
    }
    priority_queue<int, vector<int>, greater<int>> ready;  // smallest job on top
    for (int v = 1; v <= n; v++)
        if (indeg[v] == 0) ready.push(v);
    vector<int> order;
```

Listing 12.11 (Continued) *toposort.cpp: Kahn's algorithm with a min-heap of ready jobs, and cycle detection by counting.*

```cpp
    while (!ready.empty()) {
        int u = ready.top();
        ready.pop();
        order.push_back(u);
        for (int v : adj[u])
            if (--indeg[v] == 0) ready.push(v);
    }
    if ((int)order.size() < n) {  // some jobs never became ready: a cycle
        puts("Sandro fails.");
        return 0;
    }
    for (int i = 0; i < n; i++) printf("%d%c", order[i], i + 1 < n ? ' ' : '\n');
    return 0;
}
```

Listing 12.12 `toposort.py`: the same algorithm with `heapq`.

```python
# TOPOSORT - Lexicographically smallest topological order: Kahn with a min-heap.
import sys
from heapq import heapify, heappop, heappush


def main():
    data = sys.stdin.buffer.read().split()
    n, m = int(data[0]), int(data[1])
    adj = [[] for _ in range(n + 1)]
    indeg = [0] * (n + 1)
    for i in range(m):
        x, y = int(data[2 + 2 * i]), int(data[3 + 2 * i])
        adj[x].append(y)
        indeg[y] += 1
    ready = [v for v in range(1, n + 1) if indeg[v] == 0]
    heapify(ready)  # smallest job on top
    order = []
    while ready:
        u = heappop(ready)
        order.append(u)
        for v in adj[u]:
            indeg[v] -= 1
            if indeg[v] == 0:
                heappush(ready, v)
    if len(order) < n:  # some jobs never became ready: a cycle
        print("Sandro fails.")
    else:
        print(" ".join(map(str, order)))


main()
```

Listing 12.12 is the Python version; on the largest inputs it is too slow for the 0.5-second limit, as Section 12.10 measures. A plain queue instead of a heap still gives *a* topological order, just not the smallest one; on this problem that is a Wrong Answer, and on a problem that accepts any valid order (many do, with a special checker) it would be accepted. Read the output section of the statement for the words “lexicographically smallest” or “any”.

12.8.3 Running it

Our first example has six jobs. Jobs 1 and 2 are ready at the start; 3 needs 1 and 2; 5 needs 2; 4 needs 3 and 5; 6 needs 4.

```
6 6
1 3
2 3
3 4
5 4
4 6
2 5
```

```
1 2 3 5 4 6
```

Trace it by hand: the heap starts as {1,2}; taking 1 leaves job 3 waiting for 2; taking 2 frees 3 and 5; taking 3 leaves 4 waiting for 5; taking 5 frees 4; then 4 frees 6. The second example is a cycle of three rules.

```
3 3
1 2
2 3
3 1
```

```
Sandro fails.
```

12.9 Connecting Everything Cheaply: Kruskal (MST)

12.9.1 The problem in our own words

SPOJ problem MST, “Minimum Spanning Tree” [13], gives a connected undirected graph with $n \le 10{,}000$ vertices and $m \le 100{,}000$ weighted edges, with weights of absolute value at most 10^6, and asks for the total weight of a minimum spanning tree.

> **DEFINITION.** A *spanning tree* of a connected graph is a subset of its edges that forms a tree containing every vertex; it has exactly $n - 1$ edges. A *minimum spanning tree* (MST) is a spanning tree whose total weight is as small as possible.

12.9.2 Kruskal’s algorithm and why it is right

Kruskal’s algorithm [14] is greedy: sort the edges from cheapest to most expensive and go through them in that order, taking an edge whenever it joins two different pieces and skipping it when it would close a cycle. The pieces are tracked with the union-find of Section 12.3, which is why the two sections share their `DSU` code. Sorting costs $O(m \log m)$ and the union-find work is almost linear, so the whole algorithm is $O(m \log m)$.

Why is the greedy choice safe? The reason is the **cut property** [3]: take any way of splitting the vertices into two sides; among the edges that cross between the sides, a cheapest one belongs to some minimum spanning tree. Proof by exchange: take an MST that does not contain that cheapest crossing edge e. Adding e creates a cycle, and that cycle must cross the split a second time through some other edge f. Removing f gives another spanning tree whose weight changed by $w(e) - w(f) \le 0$, so it is also minimum and it contains e. When Kruskal takes an edge joining pieces A and B, it is the cheapest edge crossing between A and the rest (every cheaper edge has already been considered and either taken or found inside a piece), so the cut property says the choice is safe.

The total weight of the tree can reach $(n - 1) \times 10^6 \approx 10^{10}$, which does not fit in a 32-bit `int`. The accumulator must be a `long long`; in **Listing 12.13** that is the one line with a comment explaining why, while **Listing 12.14** needs no such care because Python integers never overflow. Negative weights are no problem for Kruskal; they simply sort first.

Listing 12.13 `mst.cpp`: Kruskal’s algorithm with union-find and a 64-bit total.

```cpp
// MST - Minimum spanning tree: Kruskal's algorithm with union-find.
#include <algorithm>
#include <cstdio>
#include <numeric>
#include <utility>
#include <vector>
using namespace std;

struct Edge {
    int w, u, v;
    bool operator<(const Edge& o) const { return w < o.w; }
};

struct DSU {
    vector<int> parent, size;
    explicit DSU(int n) : parent(n + 1), size(n + 1, 1) {
        iota(parent.begin(), parent.end(), 0);
    }
    int find(int x) {
        while (parent[x] != x) {
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }
    bool unite(int a, int b) {
        a = find(a);
        b = find(b);
        if (a == b) return false;
        if (size[a] < size[b]) swap(a, b);
        parent[b] = a;
        size[a] += size[b];
        return true;
    }
};
```

Listing 12.13 (Continued) *mst.cpp: Kruskal's algorithm with union-find and a 64-bit total.*

```cpp
int main() {
    int n, m;
    if (scanf("%d %d", &n, &m) != 2) return 0;
    vector<Edge> edges(m);
    for (auto& e : edges)
        if (scanf("%d %d %d", &e.u, &e.v, &e.w) != 3) return 0;
    sort(edges.begin(), edges.end());  // cheapest edge first
    DSU dsu(n);
    long long total = 0;  // up to (N-1) * 10^6: more than an int can hold
    int taken = 0;
    for (const auto& e : edges) {
        if (taken == n - 1) break;
        if (dsu.unite(e.u, e.v)) {  // skip edges that would close a cycle
            total += e.w;
            taken++;
        }
    }
    printf("%lld\n", total);
    return 0;
}
```

Listing 12.14 `mst.py`: the same algorithm in Python; Python integers never overflow.

```python
# MST - Minimum spanning tree: Kruskal's algorithm with union-find.
import sys


def main():
    data = sys.stdin.buffer.read().split()
    n, m = int(data[0]), int(data[1])
    edges = []
    for i in range(m):
        u, v, w = data[2 + 3 * i: 5 + 3 * i]
        edges.append((int(w), int(u), int(v)))
    edges.sort()  # cheapest edge first
    parent = list(range(n + 1))

    def find(x):
        while parent[x] != x:
            parent[x] = parent[parent[x]]
            x = parent[x]
        return x

    total = taken = 0
    for w, u, v in edges:
        if taken == n - 1:
            break
        a, b = find(u), find(v)
        if a != b:  # skip edges that would close a cycle
            parent[a] = b
            total += w
            taken += 1
    print(total)


main()
```

12.9.3 Running it

Our example has five vertices and seven edges. Sorted by weight the edges are $1-3$ (1), $2-3$ (2), $4-5$ (3), $1-2$ (4), $2-4$ (5), $3-4$ (8), $3-5$ (9). Kruskal takes the first three, skips $1-2$ because 1 and 2 are already joined through 3, takes $2-4$, and stops with four edges: $1+2+3+5=11$.

```
5 7
1 2 4
1 3 1
2 3 2
2 4 5
3 4 8
4 5 3
3 5 9
```

```
11
```

12.10 Testing the Toolkit

Every program in this chapter was checked the way Chapter 8 teaches: by comparing it with a slow program that is obviously correct, on hundreds of small random inputs. The folder `Chapter12/tests` holds three files. `gen.py` produces random inputs for each problem, either small (a handful of vertices) or of the maximum size the statement allows. `brute.py` computes the answers by methods that are too slow for the judge but hard to get wrong: a depth-first search plus an edge count for PT07Y, Floyd–Warshall all-pairs shortest paths [3] for NAKANJ, SHPATH and PT07Z, all permutations in lexicographic order for TOPOSORT, all subsets of $n-1$ edges for

MST, and all 2^n colorings for BUGLIFE. `stress.py` compiles the C++ programs and runs C++, Python and the brute force on the same inputs, stopping at the first disagreement.

The generators deliberately include the awkward cases: a single vertex, a graph with an isolated vertex, self-loops and repeated edges in PT07Y and MST, negative weights in MST, several scenarios in one BUGLIFE input, and queries whose source equals their target in NAKANJ and SHPATH. The run below used 200 random inputs per problem, with the C++ programs built with AddressSanitizer and UndefinedBehaviorSanitizer so that an out-of-bounds index would stop the run even if it happened not to change the answer.

```
buglife    200 random tests: C++ and Python agree with brute force
mst        200 random tests: C++ and Python agree with brute force
nakanj     200 random tests: C++ and Python agree with brute force
pt07y      200 random tests: C++ and Python agree with brute force
pt07z      200 random tests: C++ and Python agree with brute force
shpath     200 random tests: C++ and Python agree with brute force
toposort   200 random tests: C++ and Python agree with brute force
```

The second run times one maximum-size input per problem. The numbers are from our machine; yours will differ, but the ratios will not.

```
buglife   max input 8684.9 KB   C++  0.081 s   Python  2.930 s
mst       max input 1673.9 KB   C++  0.041 s   Python  0.665 s
nakanj    max input   24.0 KB   C++  0.046 s   Python  0.587 s
pt07y     max input   92.7 KB   C++  0.013 s   Python  0.041 s
pt07z     max input   95.5 KB   C++  0.021 s   Python  0.060 s
shpath    max input 5256.4 KB   C++  5.326 s   Python 26.867 s
toposort  max input 9549.6 KB   C++  0.100 s   Python  1.347 s
```

These timings answer the question “can Python pass?” for this chapter, with **Table 12.2** as the summary. Where the input is small (PT07Y, PT07Z, NAKANJ), Python is as good as C++. Where the input is a megabyte or more of numbers, Python spends most of its time converting text to integers and walking lists, and it is 10 to 40 times slower than C++. On TOPOSORT, whose limit is 0.5 seconds, the C++ program needs 0.10 s on the largest input we can generate and Python needs 1.35 s: submit C++.

Table 12.2 Time on our largest generated input per problem, rounded from the runs above, and whether a Python submission is realistic.

Problem	Maximum input	C++	Python	Python likely to pass?
PT07Y, PT07Z	10^4 vertices	0.01–0.02 s	0.04–0.06 s	yes
NAKANJ	4,096 queries	0.05 s	0.6 s	yes, if the limit is 1 s or more
MST	10^5 edges	0.04 s	0.7 s	borderline
BUGLIFE, TOPOSORT	10^6 edges	0.08–0.10 s	1.3–2.9 s	no
SHPATH	1,000 Dijkstra runs	5.3 s	26.9 s	no; improve the algorithm first

12.11 Using It for Real

The seven programs are templates. Most graph problems you will meet next are one of them with a different way of reading the input or a small twist, and recognizing the twist is the skill. A few examples from SPOJ, to try after this chapter:

- **A grid is a graph.** Any maze, map or board with “move to a neighboring cell” is BFS on cells, exactly like NAKANJ with four or eight moves instead of the knight’s eight. When the search

must start from many cells at once (every white pixel in a bitmap, every fire in a forest), put all of them in the queue at distance 0 before the loop starts: that is *multi-source BFS*, and SPOJ problem BITMAP [15] is the classic exercise.

- **A shortest path with costs** is SHPATH with a different input format. SPOJ problem HIGHWAYS [16] asks for the fastest route between two cities with travel times on the roads, a direct application of **Listing 12.9**.
- **A cheapest network** is MST. SPOJ problem CSTREET [17] multiplies the total length of the minimum spanning tree by a price per unit of length; the only new trap is that the product needs 64 bits.
- **An order of tasks** is topological sorting. When a problem only needs *some* order, a plain queue replaces the heap and the algorithm becomes $O(n + m)$.

Outside the judge, the same programs work on real data. The laundry van of Section 12.1 can be given a small table of places and travel minutes, typed once, and the SHPATH program answers "how soon can we be there?" for every pair. The school timetable becomes a TOPOSORT input with subjects numbered in the order the school prefers, so that the lexicographically smallest valid order is also the most natural one. The network cabling plan is an MST input with the measured cable length between each pair of rooms that can be connected. In each case the only work is to number the things, write the pairs, and read the answer back with the names.

12.12 Common Mistakes, and How to Avoid Them

Adding an undirected edge in one direction only. Every edge line of an undirected graph must produce two entries in the adjacency list. A search from vertex 1 on a one-way version of the graph reaches fewer vertices and gives answers that look plausible. The stress tests of Section 12.10 catch this at once.

Searching only from vertex 1. BUGLIFE and every "for all components" question need the outer loop over all uncolored or unvisited vertices. Small samples are often connected, so the bug survives the sample tests.

Using BFS on a weighted graph. BFS counts edges, not costs. If the edges have different weights, use Dijkstra; if the weights are only 0 and 1, a deque-based "0-1 BFS" works too (Exercise 12.7).

Using Dijkstra with negative weights. The key insight of Section 12.7.2 needs non-negative weights. With a negative edge the program may finish a vertex too early and print a distance that is too large. Negative weights call for the Bellman–Ford algorithm [3].

Forgetting the stale-entry check in lazy Dijkstra. Without `if (d != dist[u]) continue;` the program is still correct but may expand a vertex many times, and on dense inputs it can be slower by a large factor.

Recursion too deep. A recursive depth-first search on a path of 10^5 vertices can overflow the stack in C++ and certainly exceeds Python's default recursion limit. Prefer BFS, or write DFS with an explicit stack.

Overflow in the total. MST totals and path sums can exceed $2^{31} - 1$. Estimate the largest possible answer from the constraints, as Chapter 3 teaches, and choose `long long` when it exceeds about 2×10^9.

A max-heap where a min-heap is needed. `priority_queue<int>` returns the *largest* element. For Dijkstra and for the smallest ready job in TOPOSORT, pass `greater<>`, or push negated values.

Resetting per-test data. In a multi-test input, the adjacency list, the distance array and the union-find must be created fresh for each test. Declaring them inside the test loop, as the listings do, makes this automatic.

12.13 Chapter Summary

- A graph is a set of vertices and a set of edges. For judge problems store it as an adjacency list with vertices $1..n$; add undirected edges in both directions.
- **Union-find** keeps track of connected pieces under edge insertions in almost constant time per operation. A graph is a tree exactly when it has $n-1$ edges and none closes a cycle (PT07Y).
- **Breadth-first search** finds shortest paths when every edge costs the same, in $O(n+m)$ time (NAKANJ). Its variants test two-colorability (BUGLIFE) and find the diameter of a tree with two searches (PT07Z).
- **Dijkstra's algorithm** finds shortest paths with non-negative weights by always settling the closest unfinished vertex; with a binary heap and lazy deletion it runs in $O((n+m)\log n)$, Equation (12.2) (SHPATH).
- **Kahn's algorithm** produces a topological order or detects a cycle; a min-heap of ready vertices gives the lexicographically smallest order (TOPOSORT).
- **Kruskal's algorithm** builds a minimum spanning tree by taking edges from cheapest to most expensive whenever they join two pieces; the cut property proves it correct (MST).
- Every program was stress-tested against a brute force and timed on maximum inputs. C++ passes everywhere; Python is fine on small inputs and too slow where a million edges must be read.

12.14 Exercises

1. A graph has $n=6$ vertices and $m=5$ edges $\{1,2\},\{2,3\},\{3,1\},\{4,5\},\{5,6\}$. What does the PT07Y program print, and which of its two tests fails?
2. Without a computer, give the knight distance from `a1` to `b2`, and explain why it is not 2.
3. Modify the idea of BUGLIFE for a graph where some edges say "same group" and others say "different group". What does the search store on each edge, and what is the contradiction test?
4. Prove carefully that the vertex a found by the first search of Section 12.6 is an end of some longest path. Take a longest path $x-\cdots-y$ and treat separately the case where the path from s to a shares a vertex with it and the case where it does not.
5. Run Dijkstra by hand from `jember` on the map of **Figure 12.2**. List the order in which the cities are finished and the resulting distances.
6. Change **Listing 12.9** so that it computes one full distance array per distinct source in a test case and answers every query with that source from it. What is the new worst-case cost in terms of the number k of distinct sources?
7. In a graph whose edge weights are only 0 and 1, explain why a deque, with 0-edges pushed at the front and 1-edges pushed at the back, gives correct shortest distances in $O(n+m)$ time.
8. TOPOSORT with a plain FIFO queue instead of the heap: give the order it prints on our first example of Section 12.8.3 when the ready jobs are taken in the order they became ready, and say whether it is the lexicographically smallest.
9. In the MST example of Section 12.9.3, change the weight of edge $3-4$ from 8 to -2. What is the new MST weight, and which edges does it use?

10. A problem gives $n = 2 \times 10^5$ cities and $m = 4 \times 10^5$ roads and asks for one shortest path. Estimate the number of basic steps of Dijkstra with a heap, and of the $O(n^2)$ version that scans all vertices to find the closest one. Which fits in one second?

12.15 Solutions to the Exercises

1. It prints `NO`. The edge count is right, $m = 5 = n - 1$, but the edge {3,1} joins 3 and 1, which are already connected through 2, so `unite` returns false: the triangle is a cycle. (A graph with $n - 1$ edges and a cycle is always disconnected; here {4,5,6} is a separate piece.)

2. The distance is 4. A knight always moves to a square of the other color, so after an odd number of moves it stands on the other color; `a1` and `b2` are both dark, so the number of moves must be even. Two moves are not enough (from `a1` the knight reaches only `b3` and `c2`, and neither of those has a knight move to `b2`), and a four-move route exists, for example `a1`–`c2`–`e1`–`d3`–`b2`.

3. Store on each edge a parity $p \in \{0,1\}$: 0 for "same group", 1 for "different group". When the search goes from u to an uncolored v along an edge of parity p, set $\text{color}(v) = \text{color}(u) \oplus p$ (exclusive or). When it meets an already colored v, there is a contradiction exactly when $\text{color}(v) \neq \text{color}(u) \oplus p$. With $p = 1$ on every edge this is BUGLIFE.

4. Let $D = d(x, y)$ be the length of a longest path and let a be a vertex farthest from s, so $d(s, a) \geq d(s, v)$ for every v. Write P for the path $x - \cdots - y$.

Case 1: the path from s to a shares a vertex with P. Let p be the *last* vertex of the s-to-a path that lies on P, and let q be the first vertex of P reached from s. Both q and p lie on P; name the ends of P so that p lies between q and y (or $p = q$). Then the path from s to y passes through p, so $d(s, y) = d(s, p) + d(p, y)$, and $d(s, a) = d(s, p) + d(p, a)$. From $d(s, a) \geq d(s, y)$ we get $d(p, a) \geq d(p, y)$. After p the way to a leaves P for good, so walking from x along P to p and then on to a is a path, of length $d(x, p) + d(p, a) \geq d(x, p) + d(p, y) = D$. It is a longest path, and it ends at a.

Case 2: the path from s to a shares no vertex with P. Let q be the vertex of P nearest to s and let r be the last vertex that the path from s to a shares with the path from s to q; then $r \neq q$, so $d(r, q) \geq 1$. Now $d(s, y) = d(s, r) + d(r, q) + d(q, y)$ and $d(s, a) = d(s, r) + d(r, a)$, so $d(r, a) \geq d(r, q) + d(q, y)$. The path $x \to q \to r \to a$ has length $d(x, q) + d(q, r) + d(r, a) \geq d(x, q) + 2\,d(q, r) + d(q, y) = D + 2\,d(q, r) > D$, which is impossible for a longest path of length D. So Case 2 never happens, and in Case 1 the vertex a ends a longest path.

5. From `jember`: finished first `jember` (0); its neighbors give `malang` 3 and `sidoarjo` 7; `malang` (3) is finished next and improves `sidoarjo` to $3 + 2 = 5$ and `surabaya` to $3 + 4 = 7$; `sidoarjo` (5) is finished and improves `surabaya` to $5 + 1 = 6$; finally `surabaya` (6). Order: jember, malang, sidoarjo, surabaya; distances 0, 3, 5, 6. The last distance agrees with the third query of Section 12.7.4.

6. Group the queries of a test by source (a `map<int, vector<int>>` from source to query indices, keeping the original order for printing). For each distinct source run Dijkstra without the early stop and fill a full `dist` array, then answer its queries by lookup. The cost becomes $O(k\,(n + m)\log n)$ with $k \leq r$ distinct sources, instead of $O(r\,(n + m)\log n)$; when many queries share a source, k is much smaller than r.

7. The deque always holds vertices whose tentative distances are D or $D + 1$ for the current smallest value D, with all the D values at the front. A 0-edge produces a vertex at distance D,

which belongs at the front; a 1-edge produces $D + 1$, which belongs at the back. So the front of the deque is always a vertex of smallest tentative distance, which is exactly the vertex Dijkstra would take, and no heap is needed. Each vertex can be pushed a bounded number of times per improvement, and with the stale check each is expanded once: $O(n + m)$.

8. With a FIFO queue the ready jobs 1 and 2 are taken first; job 1 frees nothing; job 2 frees 3 and 5 (in that order, the order of its rules); then 3 leaves 4 waiting for 5, 5 frees 4, and 4 frees 6. The order is 1 2 3 5 4 6, which on this input happens to equal the heap order. A small change shows the difference: add a job 7 with no rules. The queue starts as [1,2,7] and prints 1 2 7 3 5 4 6; the heap prints 1 2 3 5 4 6 7. Both are valid orders, but only the heap's is lexicographically smallest, because at the third position $3 < 7$.

9. Sorted: $3 - 4$ (-2), $1 - 3$ (1), $2 - 3$ (2), $4 - 5$ (3), then the rest. Kruskal takes the first four, which already connect all five vertices: $-2 + 1 + 2 + 3 = 4$. The tree is $\{3 - 4, 1 - 3, 2 - 3, 4 - 5\}$.

10. With a heap: $(n + m)\log_2 n \approx 6 \times 10^5 \times 18 \approx 1.1 \times 10^7$ steps, far below a second. The $O(n^2)$ version needs about 4×10^{10} steps, hundreds of seconds. Only the heap version fits.

References

[1] Sphere Online Judge (SPOJ), "Problems — classical," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/classical/

[2] SPOJ, "PT07Y — Is it a tree," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/PT07Y/

[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. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, 1975.

[5] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

[6] SPOJ, "NAKANJ — Minimum Knight moves !!!," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/NAKANJ/

[7] SPOJ, "BUGLIFE — A Bug's Life," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/BUGLIFE/

[8] SPOJ, "PT07Z — Longest path in a tree," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/PT07Z/

[9] SPOJ, "SHPATH — The Shortest Path," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/SHPATH/

[10] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[11] SPOJ, "TOPOSORT — Topological Sorting," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/TOPOSORT/

[12] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[13] SPOJ, "MST — Minimum Spanning Tree," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/MST/

[14] 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, 1956.

[15] SPOJ, "BITMAP — Bitmap," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/BITMAP/

[16] SPOJ, “HIGHWAYS — Highways,” accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/HIGHWAYS/

[17] SPOJ, “CSTREET — Cobbled streets,” accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/CSTREET/

Chapter 13

Number Theory, Geometry and Strings Toolkit

Three families of problems appear on every judge again and again: problems about whole numbers (divisibility, primes, remainders), problems about points and shapes in the plane, and problems about text. Each family has a small set of techniques that turn an impossible-looking task into a few dozen lines of code, and each has its own traps: overflow in number theory, rounding in geometry, and quadratic time hidden in innocent-looking string code. This chapter builds the three toolkits around seven real SPOJ problems. By the end of it you will be able to compute huge powers modulo a number by repeated squaring (LASTDIG); tabulate Euler's totient for a million numbers with a sieve (ETF); decide whether a 63-bit number is prime with a deterministic Miller–Rabin test (PON); use the cross product for orientation, convex hulls and polygon areas with exact integers; find the closest pair among fifty thousand points by divide and conquer (CLOPPAIR); find every occurrence of a pattern in a huge text with the Knuth–Morris–Pratt algorithm (NHAY); detect repetitions with the prefix function (PERIOD); and count distinct substrings with a suffix array (DISUBSTR). As in Chapter 12, each technique comes with a proof in plain words, complete programs in C++17 and Python, real output and a stress test.

13.1 A Field, a Receipt and a Chat Log

Bu Ratna's family grows fruit and durian on a hillside plot in Jember. The plot is not a rectangle: its boundary follows a stream on one side and an old road on the other, and the land office recorded it as the GPS positions of its seven corner posts. When the family wants to know how many hectares they have, how much fertilizer to order, or how long the new fence around the whole plot must be, they need geometry: the area of a polygon given by its corners, and the length of the smallest fence that goes around a set of trees. Both are a few lines of code once you know the cross product.

Her nephew works at a shop that prints receipts with a check digit, and the shop's system must also spread its stock codes evenly over a number of shelves; both jobs are arithmetic with remainders. And her daughter, who keeps the orchard's sales in a long chat group with buyers, wants to find every message that mentions a certain buyer's order code among half a million lines of chat. Searching text quickly and correctly, even when the pattern overlaps itself, is a string problem.

Competitive programming turns these everyday needs into precise puzzles with large limits. The limits are what make the techniques necessary: a naive search is fine for one short message and hopeless for a million characters; trial division is fine for a four-digit number and hopeless for a nineteen-digit one. The three toolkits in this chapter are the standard answers, collected in textbooks such as Laaksonen's guide [1] and Halim's *Competitive Programming 4* [2], and they are the answers the SPOJ problems below expect.

13.2 The Idea in Plain Words

13.2.1 Remainders keep numbers small

Most number theory problems ask for something about numbers far too large to write down: the last digit of $7^{2147483000}$, a result "modulo $10^9 + 7$", whether a nineteen-digit number is prime. The key tool is **modular arithmetic**. For a positive integer m, $a \bmod m$ is the remainder when a is divided by m, a number between 0 and $m - 1$. Remainders behave well under addition and multiplication:

$$(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m \tag{13.1}$$

and the same holds with $+$ in place of $\cdot$. So we may reduce after every step and never hold a number larger than about m^2. With m below 2^{31} the product fits in a 64-bit `long long`; with m up to 2^{63} it does not, and we need a 128-bit product, as Section 13.5 shows.

13.2.2 Exact integers for geometry

Geometry problems give coordinates, and the tempting approach is to compute angles and slopes with `double`. Most of the time there is a better way: almost every basic question (does the path turn left or right, do two segments cross, is a point inside a triangle, what is the area of a polygon) can be answered with one quantity, the **cross product**, which uses only multiplication and subtraction. With integer coordinates it is computed exactly, and the program never has to decide whether 10^{-12} is "zero". Floating point appears only at the very end, when the answer itself is a distance with six decimals.

13.2.3 Never look at the same character twice

The naive way to find a pattern of length m in a text of length n compares the pattern at every position: up to $n \cdot m$ character comparisons, 10^{12} for a million-character text and pattern. The string algorithms of this chapter share one idea: when a comparison fails, use what you have already learned about the text instead of starting over. The prefix function (Section 13.8) records, for every prefix of the pattern, how much of it can be reused, and that brings the search down to $O(n + m)$.

Table 13.1 lists the problems of the chapter with the constraint that forces each technique.

Table 13.1 The problems of this chapter, the limit that rules out the naive approach, and the technique that fits.

Section	SPOJ problem	Constraint that decides	Technique	Time
13.3	LASTDIG	$b \le 2.1 \times 10^9$	fast exponentiation	$O(\log b)$
13.4	ETF	20,000 queries, $n \le 10^6$	totient sieve	$O(N \log\log N)$ once
13.5	PON	$n < 2^{63}$	Miller–Rabin, 12 bases	$O(\log n)$ products
13.6	—	exact answers	cross product, hull, shoelace	$O(n \log n)$
13.7	CLOPPAIR	$n \le 50{,}000$	divide and conquer	$O(n \log n)$
13.8	NHAY	text of unbounded length	Knuth–Morris–Pratt	$O(n + m)$
13.9	PERIOD	$n \le 10^6$	prefix function	$O(n)$
13.10	DISUBSTR	$n \le 1{,}000$	suffix array + LCP	$O(n \log^2 n)$

13.3 Huge Powers: Fast Exponentiation (LASTDIG)

13.3.1 The problem in our own words

SPOJ problem LASTDIG, "The last digit" [3], gives up to 30 pairs of integers a and b with $0 \le a \le 20$ and $0 \le b \le 2{,}147{,}483{,}000$, never both zero, and asks for the last digit of a^b. The number a^b itself can have billions of digits, so computing it and taking the last digit is out of the question, even with Python's unlimited integers.

13.3.2 The idea: square and multiply

The last digit of a number is its remainder modulo 10, and by Equation (13.1) we can reduce modulo 10 after every multiplication. That alone is not enough: multiplying b times costs two billion steps. **Fast exponentiation** (binary exponentiation) uses the binary digits of b. Write $b = \sum_k \beta_k 2^k$ with each bit $\beta_k \in \{0,1\}$. Then

$$a^b = \prod_{k\,:\,\beta_k = 1} a^{2^k}, \qquad a^{2^{k+1}} = (a^{2^k})^2 \tag{13.2}$$

so the program keeps a running square $a, a^2, a^4, a^8, \ldots$, one squaring per bit, and multiplies the result by the current square whenever the bit is 1. The number of steps is the number of bits of b, at most 31 here; **Listing 13.1** and **Listing 13.2** implement it. The same routine, with a different modulus, is the workhorse of Section 13.5 and of countless problems that say "print the answer modulo $10^9 + 7$".

Listing 13.1 `lastdig.cpp`: fast exponentiation modulo 10.

```cpp
// LASTDIG - The last digit of a^b: fast exponentiation modulo 10.
#include <cstdio>

long long powMod(long long base, long long exp, long long mod) {
    long long result = 1 % mod;
    base %= mod;
    while (exp > 0) {
        if (exp & 1) result = result * base % mod;  // this bit of exp is set
        base = base * base % mod;                   // base^(2^k) for the next bit
        exp >>= 1;
    }
    return result;
}

int main() {
    int t;
    if (scanf("%d", &t) != 1) return 0;
    while (t--) {
        long long a, b;
        if (scanf("%lld %lld", &a, &b) != 2) return 0;
        printf("%lld\n", powMod(a, b, 10));
    }
    return 0;
}
```

Listing 13.2 `lastdig.py`: the same loop in Python; the built-in `pow(a, b, 10)` does the same in one call.

```python
# LASTDIG - The last digit of a^b: fast exponentiation modulo 10.
import sys


def pow_mod(base, exp, mod):
    result = 1 % mod
    base %= mod
    while exp > 0:
        if exp & 1:
            result = result * base % mod  # this bit of exp is set
        base = base * base % mod  # base^(2^k) for the next bit
        exp >>= 1
    return result


def main():
    data = sys.stdin.read().split()
    t = int(data[0])
    out = [pow_mod(int(data[1 + 2 * i]), int(data[2 + 2 * i]), 10)
           for i in range(t)]
    print("\n".join(map(str, out)))  # pow(a, b, 10) is the built-in equivalent


main()
```

Two edge cases decide this problem. When $b = 0$ the answer is 1 (for $a \neq 0$), and the program must print it; a loop that starts with `result = base` instead of `result = 1` gets it wrong. And when $a = 0$ and $b > 0$ the answer is 0. The expression `1 % mod` at the start makes the routine correct even for $m = 1$, where every remainder is 0, a case that costs nothing to handle and appears in other problems.

13.3.3 Running it

```
4
2 10
3 0
0 7
7 2147483000
```

```
4
1
0
1
```

$2^{10} = 1024$ ends in 4; $3^0 = 1$; $0^7 = 0$; and the powers of 7 end in the repeating cycle 7,9,3,1 of length 4, and 2,147,483,000 is divisible by 4, so $7^{2147483000}$ ends in 1. That cycle gives a second solution: every last digit repeats with period 1, 2 or 4, so a^b mod 10 depends only on a mod 10 and on b mod 4 (with $b = 0$ handled separately). Fast exponentiation needs no such insight and works for every modulus, which is why it is the tool to learn.

13.4 Counting Coprimes for a Million Numbers: Euler's Totient (ETF)

13.4.1 The problem in our own words

SPOJ problem ETF, "Euler Totient Function" [4], gives up to 20,000 numbers n with $1 \leq n \leq 10^6$ and asks, for each, how many integers k between 1 and n have no common divisor with n other than 1.

DEFINITION. Two integers are *coprime* when their greatest common divisor is 1. *Euler's totient* $\varphi(n)$ is the number of integers k with $1 \le k \le n$ that are coprime to n. For example $\varphi(9) = 6$: the numbers 1, 2, 4, 5, 7, 8.

13.4.2 The product formula

Counting with `gcd` for every k costs $O(n \log n)$ per query, $2 \times 10^4 \times 10^6$ steps in the worst case: far too slow. The formula that makes φ cheap is

$$\varphi(n) = n \prod_{p \mid n} \left(1 - \frac{1}{p}\right) \tag{13.3}$$

where the product runs over the distinct primes p that divide n [5]. The reason, in plain words: among the numbers $1..n$, exactly a fraction $1/p$ are multiples of p, and for different primes these fractions combine independently (this is the inclusion–exclusion principle, or the Chinese remainder theorem, underneath). For $n = 36 = 2^2 \cdot 3^2$: $\varphi(36) = 36 \cdot \frac{1}{2} \cdot \frac{2}{3} = 12$.

Equation (13.3) suggests two programs. With many queries and a small upper limit, compute φ for *every* number up to 10^6 at once with a sieve: start with $\varphi(i) = i$, and for every prime p multiply the entries of all its multiples by $1 - 1/p$, which in integers is `phi[j] -= phi[j] / p`. A number p is prime exactly when no smaller prime has touched it, that is, when `phi[p]` still equals p. The sieve costs $\sum_p N/p = O(N \log\log N)$ steps, about three million for $N = 10^6$, and then every query is a table lookup. That is **Listing 13.3**.

Listing 13.3 `etf.cpp`: the totient sieve; each query is answered from the table.

```
// ETF - Euler's totient for many n <= 10^6: one sieve, then O(1) per query.
#include <cstdio>
#include <vector>
using namespace std;

int main() {
    const int LIMIT = 1000000;
    vector<int> phi(LIMIT + 1);
    for (int i = 0; i <= LIMIT; i++) phi[i] = i;
    for (int p = 2; p <= LIMIT; p++) {
        if (phi[p] != p) continue;  // p was touched by a smaller prime: composite
        for (int j = p; j <= LIMIT; j += p)
            phi[j] -= phi[j] / p;  // multiply by (1 - 1/p)
    }
    int t;
    if (scanf("%d", &t) != 1) return 0;
    while (t--) {
        int n;
        if (scanf("%d", &n) != 1) return 0;
        printf("%d\n", phi[n]);
    }
    return 0;
}
```

In Python a sieve over a million entries with a Python-level inner loop takes a few seconds, so **Listing 13.4** takes the other road: it factorizes each query by trial division with the 168 primes below 1,000. Any $n \le 10^6$ has at most one prime factor above $\sqrt{n} \le 1000$, and whatever is left after dividing out the small primes is that factor. The cost is at most 168 divisions per query, about 3.4 million in total, which Python handles in a fraction of a second.

Listing 13.4 `etf.py`: $\varphi(n)$ by factorization with the primes below 1,000.

```python
# ETF - Euler's totient for many n <= 10^6: factorize n with the primes below 1000.
import sys

PRIMES = [p for p in range(2, 1001)
          if all(p % d for d in range(2, int(p ** 0.5) + 1))]


def phi(n):
    result = n
    for p in PRIMES:
        if p * p > n:
            break
        if n % p == 0:
            while n % p == 0:
                n //= p
            result -= result // p  # multiply by (1 - 1/p)
    if n > 1:  # what is left is one prime factor above sqrt(original n)
        result -= result // n
    return result


def main():
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    print("\n".join(str(phi(int(x))) for x in data[1:1 + t]))


main()
```

13.4.3 Running it

```
5
1
9
10
36
999983
```

```
1
6
4
12
999982
```

$\varphi(1) = 1$ by definition (1 is coprime to itself); $\varphi(9) = 9 \cdot \frac{2}{3} = 6$; $\varphi(10) = 10 \cdot \frac{1}{2} \cdot \frac{4}{5} = 4$; $\varphi(36) = 12$ as computed above; and 999,983 is prime, so $\varphi = 999{,}982$.

13.5 Is a 19-Digit Number Prime? Miller–Rabin (PON)

13.5.1 The problem in our own words

SPOJ problem PON, "Prime or Not" [6], gives up to 500 integers n with $2 \le n \le 2^{63} - 1$ and asks `YES` or `NO`: is n prime? Trial division up to $\sqrt{n} \approx 3 \times 10^9$ is hopeless, and a sieve even more so.

13.5.2 From Fermat to Miller–Rabin

Fermat's little theorem says that for a prime p and any a not divisible by p,

$$a^{p-1} \equiv 1 \pmod{p}. \tag{13.4}$$

So if $a^{n-1} \bmod n \neq 1$ for some a, then n is certainly composite, and fast exponentiation computes that power in about 63 squarings. Unfortunately the converse fails: there are composite numbers, the *Carmichael numbers*, for which $a^{n-1} \equiv 1$ for every a coprime to n. The smallest is $561 = 3 \cdot 11 \cdot 17$.

The **Miller–Rabin** test [7] strengthens Fermat's test with one more fact: modulo a prime, the only square roots of 1 are 1 and -1 (that is, $n-1$). Write $n - 1 = d \cdot 2^s$ with d odd. For a prime n, the sequence

$$a^d,\ a^{2d},\ a^{4d},\ \ldots,\ a^{2^s d} = a^{n-1} \pmod{n} \tag{13.5}$$

ends in 1 by Equation (13.4), and each term is the square of the previous one, so the first 1 in the sequence must be preceded by $n-1$, unless the sequence starts with 1. A base a for which this fails is a *witness* that n is composite. For a composite n at least three quarters of all bases are witnesses, which makes a randomized test very reliable; but a judge wants certainty, and for 64-bit numbers certainty is available. It has been verified by exhaustive computation that the first twelve primes 2,3,5, … ,37 used as bases give no false answer for any n below 3.18×10^{23} [8], far above $2^{63} \approx 9.2 \times 10^{18}$. So the test with those twelve bases is a *deterministic* primality test for every input of PON.

13.5.3 The 128-bit product

Squaring a remainder close to 2^{63} gives a product close to 2^{126}, which overflows 64 bits and silently gives garbage. GCC and Clang provide the type `unsigned __int128`, whose products are exact; **Listing 13.5** uses it in `mulMod`. It is a compiler extension, not standard C++, and Microsoft's compiler does not have it (Chapter 5 discusses such portability questions); on SPOJ, which compiles C++ with GCC, it is available. Python's integers have no size limit, so **Listing 13.6** uses the built-in three-argument `pow`.

Listing 13.5 `pon.cpp`: deterministic Miller–Rabin with twelve prime bases and 128-bit products.

```cpp
// PON - Prime or not for n < 2^63: deterministic Miller-Rabin.
#include <cstdint>
#include <cstdio>

typedef uint64_t u64;
typedef unsigned __int128 u128;  // a GCC/Clang extension, not standard C++

u64 mulMod(u64 a, u64 b, u64 m) { return (u64)((u128)a * b % m); }

u64 powMod(u64 a, u64 e, u64 m) {
    u64 r = 1;
    a %= m;
    for (; e > 0; e >>= 1) {
        if (e & 1) r = mulMod(r, a, m);
        a = mulMod(a, a, m);
    }
    return r;
}

bool isPrime(u64 n) {
    if (n < 2) return false;
    const u64 bases[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
    for (u64 p : bases)
        if (n % p == 0) return n == p;  // small primes and their multiples
    u64 d = n - 1;
    int s = 0;
    while ((d & 1) == 0) {  // n - 1 = d * 2^s with d odd
        d >>= 1;
        s++;
    }
    for (u64 a : bases) {
        u64 x = powMod(a, d, n);
        if (x == 1 || x == n - 1) continue;
        bool witness = true;  // a proves n composite unless we reach n - 1
        for (int r = 1; r < s && witness; r++) {
            x = mulMod(x, x, n);
            if (x == n - 1) witness = false;
        }
        if (witness) return false;
    }
    return true;  // these 12 bases suffice for every n below 3.18 * 10^23
}
```

Listing 13.5 (Continued) *pon.cpp: deterministic Miller–Rabin with twelve prime bases and 128-bit products.*

```cpp
int main() {
    int t;
    if (scanf("%d", &t) != 1) return 0;
    while (t--) {
        unsigned long long n;
        if (scanf("%llu", &n) != 1) return 0;
        puts(isPrime(n) ? "YES" : "NO");
    }
    return 0;
}
```

Listing 13.6 `pon.py`: the same test in Python.

```python
# PON - Prime or not for n < 2^63: deterministic Miller-Rabin.
import sys

BASES = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37)


def is_prime(n):
    if n < 2:
        return False
    for p in BASES:
        if n % p == 0:
            return n == p  # small primes and their multiples
    d, s = n - 1, 0
    while d % 2 == 0:  # n - 1 = d * 2^s with d odd
        d //= 2
        s += 1
    for a in BASES:
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(s - 1):
            x = x * x % n
            if x == n - 1:
                break
        else:
            return False  # a is a witness: n is composite
    return True  # these 12 bases suffice for every n below 3.18 * 10^23


def main():
    data = sys.stdin.read().split()
    t = int(data[0])
    print("\n".join("YES" if is_prime(int(x)) else "NO" for x in data[1:1 + t]))


main()
```

The small-prime loop at the start of `isPrime` does two jobs: it answers directly for the twelve base primes themselves (a base must not be a multiple of n), and it throws out the multiples of those primes, which are the vast majority of composites, before any exponentiation. The `for ... else` in Python runs the `else` part only when the loop was not left by `break`, which is exactly "no square step reached $n-1$".

13.5.4 Running it

```
6
2
561
1000000007
3825123056546413051
9223372036854775783
9223372036854775807
```

```
YES
NO
YES
NO
YES
NO
```

The inputs are chosen to be hard: 561 is the smallest Carmichael number; 10^9+7 is the prime that appears in half the "modulo" statements of the world; 3,825,123,056,546,413,051 is a composite that passes the Miller–Rabin test for every prime base up to 31 and is caught only by

the twelfth base, 37; $2^{63} - 25$ is the largest prime below 2^{63}; and $2^{63} - 1 = 7^2 \cdot 73 \cdot 127 \cdot 337 \cdot 92737 \cdot 649657$ is composite.

13.6 A Geometry Kit: Orientation, Convex Hull and Area

13.6.1 The cross product

For three points a, b, c in the plane, the **cross product** of the vectors $b - a$ and $c - a$ is

$$\operatorname{cross}(a, b, c) = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x). \tag{13.6}$$

Its sign tells the *orientation*: positive when the path $a \to b \to c$ turns left (counterclockwise), negative when it turns right, zero when the three points lie on one line. Its absolute value is twice the area of the triangle abc [9]. With integer coordinates up to 10^6 in absolute value, each product is at most about 4×10^{12}, well inside a `long long`. Almost every geometric predicate is built from Equation (13.6): two segments cross when the ends of each lie on different sides of the other, a point is inside a convex polygon when it is on the left of every edge, and so on.

13.6.2 The convex hull

The *convex hull* of a set of points is the smallest convex polygon that contains them all, the shape a rubber band takes when it is stretched around nails at the points. For the orchard of Section 13.1 it is the shortest fence that encloses every tree. **Andrew's monotone chain** algorithm [10] builds it in $O(n \log n)$: sort the points by x (then y), sweep from left to right keeping the *lower* hull as a stack, and pop the last point while it and the new point make a turn that is not to the left; then sweep from right to left for the *upper* hull. Every point is pushed and popped at most once per sweep, so after sorting the work is linear. With the test `cross(...) <= 0`, points that lie on a hull edge are dropped; with `< 0` they would be kept, and some problems ask for one, some for the other.

13.6.3 The area of a polygon

The area of a simple polygon with corners $p_0, p_1, \dots, p_{k-1}$ in order is given by the **shoelace formula** [9]:

$$A = \frac{1}{2}\left|\sum_{i=0}^{k-1} (x_i\, y_{i+1} - y_i\, x_{i+1})\right|, \qquad p_k = p_0. \tag{13.7}$$

Each term is the cross product of two consecutive corners seen from the origin, twice the signed area of a triangle; the triangles outside the polygon cancel. The sum inside the bars is an integer for integer corners, so the program keeps *twice* the area as an integer and prints the half at the end, with `.0` or `.5`: no rounding anywhere.

Listing 13.7 (**Listing 13.8** in Python) puts the three tools together and runs them on thirteen points; **Figure 13.1** shows the result. The point (0,2) lies on the edge between (0,0) and (0,4) and is dropped, and the five points inside the hull are ignored.

Listing 13.7 `geometry.cpp`: orientation by cross product, the monotone-chain convex hull, and the shoelace area.

```cpp
// A small geometry kit: orientation, convex hull (monotone chain), polygon area.
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

struct Pt {
    long long x, y;
    bool operator<(const Pt& o) const { return x < o.x || (x == o.x && y < o.y); }
};

// Cross product of (b - a) and (c - a): > 0 left turn, < 0 right, 0 collinear.
long long cross(const Pt& a, const Pt& b, const Pt& c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

// Andrew's monotone chain: counterclockwise hull, collinear points dropped.
vector<Pt> convexHull(vector<Pt> p) {
    sort(p.begin(), p.end());
    p.erase(unique(p.begin(), p.end(), [](const Pt& a, const Pt& b) {
        return a.x == b.x && a.y == b.y;
    }), p.end());
    if (p.size() < 3) return p;
    vector<Pt> h(2 * p.size());
    size_t k = 0;
    for (size_t i = 0; i < p.size(); i++) {  // lower hull, left to right
        while (k >= 2 && cross(h[k - 2], h[k - 1], p[i]) <= 0) k--;
        h[k++] = p[i];
    }
    for (size_t i = p.size() - 1, t = k + 1; i-- > 0;) {  // upper hull, back
        while (k >= t && cross(h[k - 2], h[k - 1], p[i]) <= 0) k--;
        h[k++] = p[i];
    }
    h.resize(k - 1);  // the last point repeats the first
    return h;
}
```

Listing 13.7 (Continued) *geometry.cpp: orientation by cross product, the monotone-chain convex hull, and the shoelace area.*

```
// Shoelace formula: twice the signed area (positive when counterclockwise).
long long twiceArea(const vector<Pt>& poly) {
    long long s = 0;
    for (size_t i = 0; i < poly.size(); i++) {
        const Pt& a = poly[i];
        const Pt& b = poly[(i + 1) % poly.size()];
        s += a.x * b.y - a.y * b.x;
    }
    return s;
}

int main() {
    Pt a{0, 0}, b{4, 0}, c{2, 3}, d{6, 0};
    printf("cross(a,b,c) = %lld (left turn)\n", cross(a, b, c));
    printf("cross(a,c,b) = %lld (right turn)\n", cross(a, c, b));
    printf("cross(a,b,d) = %lld (collinear)\n", cross(a, b, d));
    vector<Pt> pts = {{0, 0}, {4, -1}, {8, 1}, {8, 3}, {7, 5}, {3, 6}, {0, 4},
                      {0, 2}, {4, 2}, {2, 3}, {5, 3}, {6, 1}, {1, 1}};
    vector<Pt> hull = convexHull(pts);
    printf("hull (%zu vertices):", hull.size());
    for (const Pt& p : hull) printf(" (%lld,%lld)", p.x, p.y);
    long long a2 = twiceArea(hull);
    printf("\ntwice the area = %lld, area = %lld.%s\n", a2, a2 / 2,
           a2 % 2 ? "5" : "0");
    return 0;
}
```

Listing 13.8 `geometry.py`: the same kit in Python.

```python
# A small geometry kit: orientation, convex hull (monotone chain), polygon area.


def cross(a, b, c):
    """Cross product of (b - a) and (c - a): > 0 left, < 0 right, 0 collinear."""
    return (b[0] - a[0]) * (c[1] - a[1]) - (b[1] - a[1]) * (c[0] - a[0])


def convex_hull(points):
    """Andrew's monotone chain: counterclockwise, collinear points dropped."""
    p = sorted(set(points))
    if len(p) < 3:
        return p
    lower, upper = [], []
    for q in p:  # lower hull, left to right
        while len(lower) >= 2 and cross(lower[-2], lower[-1], q) <= 0:
            lower.pop()
        lower.append(q)
    for q in reversed(p):  # upper hull, right to left
        while len(upper) >= 2 and cross(upper[-2], upper[-1], q) <= 0:
            upper.pop()
        upper.append(q)
    return lower[:-1] + upper[:-1]  # each chain ends where the other starts


def twice_area(poly):
    """Shoelace formula: twice the signed area (positive if counterclockwise)."""
    n = len(poly)
    return sum(poly[i][0] * poly[(i + 1) % n][1] - poly[i][1] * poly[(i + 1) % n][0]
               for i in range(n))


def main():
    a, b, c, d = (0, 0), (4, 0), (2, 3), (6, 0)
    print("cross(a,b,c) = %d (left turn)" % cross(a, b, c))
    print("cross(a,c,b) = %d (right turn)" % cross(a, c, b))
    print("cross(a,b,d) = %d (collinear)" % cross(a, b, d))
    pts = [(0, 0), (4, -1), (8, 1), (8, 3), (7, 5), (3, 6), (0, 4),
           (0, 2), (4, 2), (2, 3), (5, 3), (6, 1), (1, 1)]
    hull = convex_hull(pts)
    print("hull (%d vertices):" % len(hull), " ".join("(%d,%d)" % p for p in hull))
    a2 = twice_area(hull)
    half = "5" if a2 % 2 else "0"
    print("twice the area = %d, area = %d.%s" % (a2, a2 // 2, half))
```

Listing 13.8 (Continued) *geometry.py: the same kit in Python.*

```python
main()
```

```
cross(a,b,c) = 12 (left turn)
cross(a,c,b) = -12 (right turn)
cross(a,b,d) = 0 (collinear)
hull (7 vertices): (0,0) (4,-1) (8,1) (8,3) (7,5) (3,6) (0,4)
twice the area = 86, area = 43.0
```

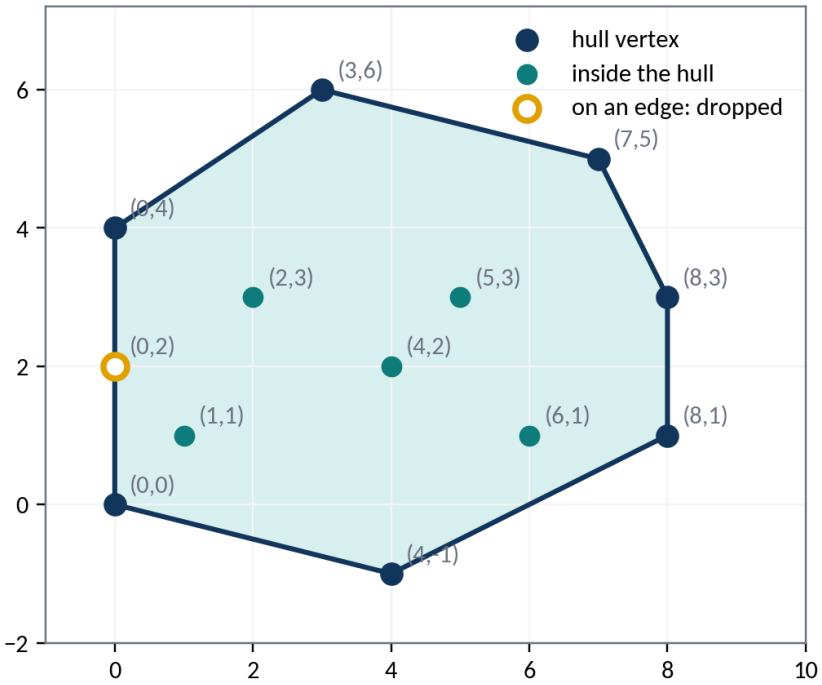


Figure 13.1 The thirteen points of **Listing 13.7**: seven hull vertices, five interior points, and one point on a hull edge that the monotone chain drops.

The hull has seven vertices, listed counterclockwise starting from the lowest-leftmost point, and its area is 43 square units. For Bu Ratna's plot, the seven corner posts in meters give the area in square meters by the same formula; dividing by 10,000 gives hectares. The real work in such an application is converting GPS latitude and longitude into meters on a local flat grid first, which for a plot a few hundred meters wide can be done with a simple scale factor per degree.

13.7 The Closest Pair of Points (CLOPPAIR)

13.7.1 The problem in our own words

SPOJ problem CLOPPAIR, "Closest Point Pair" [11], gives $n \le 50{,}000$ distinct points with integer coordinates of absolute value at most 10^6, and promises that exactly one pair has the smallest distance. The program must print the indices of that pair (counted from 0, smaller index first) and the distance, rounded to six decimals.

13.7.2 Divide and conquer

Comparing all pairs costs $n(n-1)/2 \approx 1.25 \times 10^9$ distance computations: too slow. The classic $O(n \log n)$ algorithm [12], [13] is divide and conquer. Sort the points by x and split them at the median into a left half and a right half. Recursively find the closest pair in each half; let d be the smaller of the two distances. The closest pair overall is either one of those two pairs, or a pair with one point on each side of the dividing line. Such a pair can only be closer than d if both points lie in the *strip* of width $2d$ around the line, the shaded band in **Figure 13.2**.

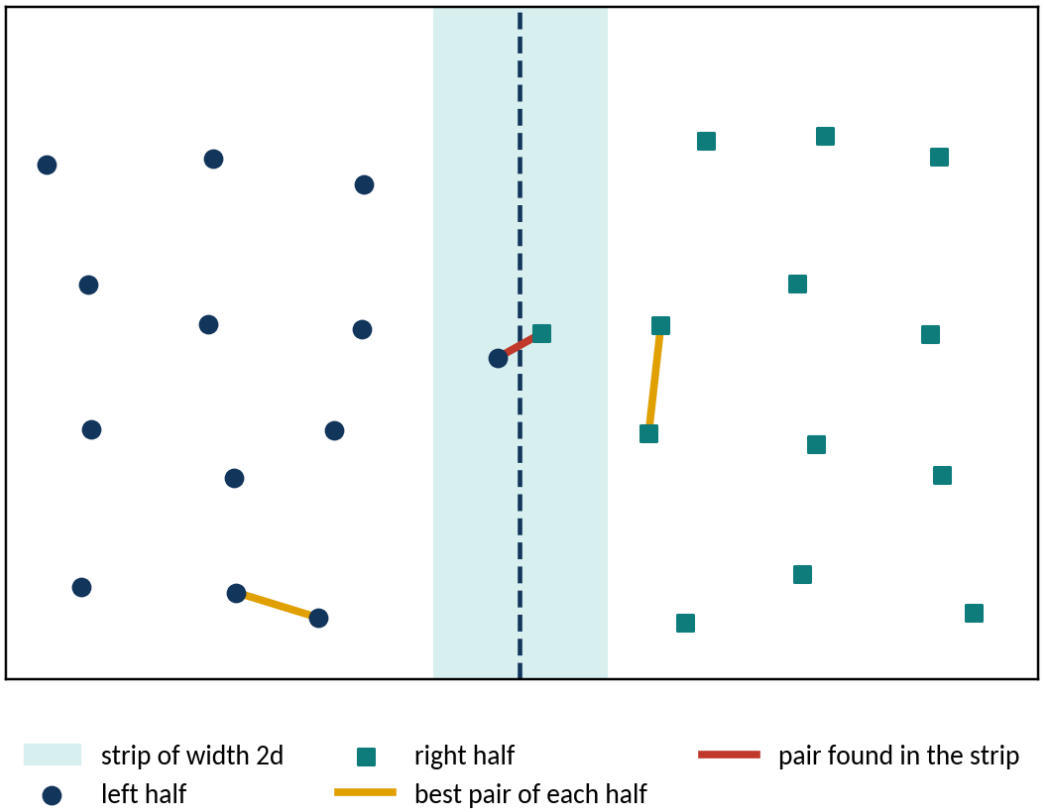


Figure 13.2 The combine step of the closest-pair algorithm: the best pair of each half gives d, and only points in the strip of width $2d$ around the dividing line need to be compared across it.

The crucial fact is that within the strip each point needs to be compared with only a constant number of others. Sort the strip points by y. For a point p, any point q of the strip that is closer than d must also be within d vertically. The rectangle of size $2d \times d$ above p can hold at most 8 points, because each of its two $d \times d$ squares lies in one half and points in the same half are at least d apart, so at most 4 fit in each square. Therefore the inner loop, which walks back from p through the strip in order of y and stops as soon as the vertical gap reaches d, looks at a constant number of points. The strip step is linear, and the running time satisfies $T(n) = 2T(n/2) + O(n)$, which solves to $O(n\log n)$ (a recurrence of the kind Chapter 3 budgets for).

Two implementation details make **Listing 13.9** and **Listing 13.10** fast and simple. The recursion returns its points sorted by y, by merging the two sorted halves, exactly as merge sort does; so no sorting happens inside the recursion. And all comparisons use the *squared* distance, an exact integer up to 8×10^{12}; the square root is taken once, at the very end, for printing.

Listing 13.9 `cloppair.cpp`: divide and conquer on squared integer distances; each call leaves its part of the array sorted by y.

```cpp
// CLOPPAIR - Closest pair of points: divide and conquer in O(n log n).
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <vector>
using namespace std;

struct Pt {
    long long x, y;
    int id;
};

long long bestD2 = -1;  // squared distance of the best pair so far (-1: none)
int bestA = -1, bestB = -1;

void consider(const Pt& p, const Pt& q) {
    long long dx = p.x - q.x, dy = p.y - q.y, d2 = dx * dx + dy * dy;
    if (bestD2 < 0 || d2 < bestD2) {
        bestD2 = d2;
        bestA = min(p.id, q.id);
        bestB = max(p.id, q.id);
    }
}

bool byY(const Pt& a, const Pt& b) { return a.y < b.y; }

// On entry pts[lo, hi) is sorted by x; on exit it is sorted by y.
void solve(vector<Pt>& pts, int lo, int hi, vector<Pt>& buf) {
    if (hi - lo <= 3) {
        for (int i = lo; i < hi; i++)
            for (int j = i + 1; j < hi; j++) consider(pts[i], pts[j]);
        sort(pts.begin() + lo, pts.begin() + hi, byY);
        return;
    }
    int mid = (lo + hi) / 2;
    long long midX = pts[mid].x;
    solve(pts, lo, mid, buf);
    solve(pts, mid, hi, buf);
    merge(pts.begin() + lo, pts.begin() + mid, pts.begin() + mid,
          pts.begin() + hi, buf.begin(), byY);
    copy(buf.begin(), buf.begin() + (hi - lo), pts.begin() + lo);
    int stripLen = 0;  // the strip lives in buf, sorted by y
    for (int i = lo; i < hi; i++) {
        long long dx = pts[i].x - midX;
        if (dx * dx >= bestD2) continue;  // too far from the dividing line
        for (int j = stripLen - 1; j >= 0; j--) {
```

Listing 13.9 (Continued) *cloppair.cpp: divide and conquer on squared integer distances; each call leaves its part of the array sorted by y.*

```
            long long dy = pts[i].y - buf[j].y;
            if (dy * dy >= bestD2) break;  // only a few j pass this test
            consider(pts[i], buf[j]);
        }
        buf[stripLen++] = pts[i];
    }
}

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    vector<Pt> pts(n), buf(n);
    for (int i = 0; i < n; i++) {
        if (scanf("%lld %lld", &pts[i].x, &pts[i].y) != 2) return 0;
        pts[i].id = i;
    }
    sort(pts.begin(), pts.end(), [](const Pt& a, const Pt& b) {
        return a.x < b.x || (a.x == b.x && a.y < b.y);
    });
    solve(pts, 0, n, buf);
    printf("%d %d %.6f\n", bestA, bestB, sqrt((double)bestD2));
    return 0;
}
```

Listing 13.10 `cloppair.py`: the same algorithm in Python, returning a new list sorted by y from each call.

```python
# CLOPPAIR - Closest pair of points: divide and conquer in O(n log n).
import sys
from heapq import merge
from math import sqrt


def closest(pts):
    """pts sorted by x; returns ((d2, a, b), pts sorted by y)."""
    n = len(pts)
    best = (float("inf"), -1, -1)
    if n <= 3:
        for i in range(n):
            for j in range(i + 1, n):
                (x1, y1, a), (x2, y2, b) = pts[i], pts[j]
                d2 = (x1 - x2) ** 2 + (y1 - y2) ** 2
                best = min(best, (d2, min(a, b), max(a, b)))
        return best, sorted(pts, key=lambda p: p[1])
    mid = n // 2
    mid_x = pts[mid][0]
    left, ly = closest(pts[:mid])
    right, ry = closest(pts[mid:])
    best = min(best, left, right)
    by_y = list(merge(ly, ry, key=lambda p: p[1]))
    strip = []  # points near the dividing line, sorted by y
    for p in by_y:
        if (p[0] - mid_x) ** 2 >= best[0]:
            continue  # too far from the dividing line
        for q in reversed(strip):
            if (p[1] - q[1]) ** 2 >= best[0]:
                break  # only a few q pass this test
            d2 = (p[0] - q[0]) ** 2 + (p[1] - q[1]) ** 2
            best = min(best, (d2, min(p[2], q[2]), max(p[2], q[2])))
        strip.append(p)
    return best, by_y


def main():
    data = sys.stdin.buffer.read().split()
    n = int(data[0])
    pts = sorted((int(data[1 + 2 * i]), int(data[2 + 2 * i]), i) for i in range(n))
    (d2, a, b), _ = closest(pts)
    print("%d %d %.6f" % (a, b, sqrt(d2)))


main()
```

The points carry their original index through the sorting, and the pair is stored as (smaller index, larger index), because the output demands $a < b$ and the recursion may meet the two points in either order.

13.7.3 Running it

```
6
0 0
5 5
-3 4
6 7
10 -2
1 9
```

```
1 3 2.236068
```

Points 1 and 3 are (5,5) and (6,7), at distance $\sqrt{1+4} = \sqrt{5} \approx 2.236068$. The format string `%.6f` prints exactly six decimals, which is what the statement asks; `cout << d` without `fixed` and `setprecision(6)` would print `2.23607`, a Wrong Answer on a judge that compares text.

13.8 Finding a Needle in a Haystack: Knuth–Morris–Pratt (NHAY)

13.8.1 The problem in our own words

SPOJ problem NHAY, "A Needle in the Haystack" [14], gives several test cases, until the end of the input. Each gives the length of a *needle* (a pattern), the needle, and a *haystack* (a text) whose length the statement deliberately leaves unbounded, with a remark that the program should not store the whole haystack. For each test the program prints every position (counted from 0) where the needle occurs in the haystack, one per line. SPOJ's default checker compares the output token by token and ignores extra blank lines, so the empty line our programs print after each test case is harmless.

13.8.2 Borders and the prefix function

> **DEFINITION.** A *border* of a string s is a string that is both a proper prefix and a suffix of s. For example, `aba` has the borders `a` and the empty string; `abab` has the border `ab`. The *prefix function* of a string s of length m is the array $\pi[0..m-1]$ in which $\pi[i]$ is the length of the longest border of the prefix $s[0..i]$.

For the needle `aabaab`, the prefix function is $\pi = [0,1,0,1,2,3]$: the prefix `aabaab` ends with `aab`, which is also how it starts. **Table 13.2** computes it step by step.

Table 13.2 The prefix function of `aabaab`, one prefix per row.

i	Prefix $s[0..i]$	Longest border	$\pi[i]$
0	`a`	(none)	0
1	`aa`	`a`	1
2	`aab`	(none)	0
3	`aaba`	`a`	1
4	`aabaa`	`aa`	2
5	`aabaab`	`aab`	3

The prefix function can be computed in $O(m)$ time by reusing earlier values [12], [15]. To extend a border of $s[0..i-1]$ of length $k = \pi[i-1]$, compare $s[i]$ with $s[k]$, the character that follows that border. If they match, the border grows to $k+1$. If not, try the next shorter border, which is the longest border of the border itself, of length $\pi[k-1]$, and repeat; when k reaches 0 and still no match, $\pi[i] = 0$.

$$\pi[i] = \max\{\, k+1 \,:\, k \in \{\pi[i-1],\ \pi[\pi[i-1]-1],\ \ldots,\ 0\},\ s[k] = s[i] \,\} \cup \{0\} \tag{13.8}$$

Why is this linear and not quadratic, given the inner `while` loop? The value k goes up by at most one per character and every step of the `while` loop lowers it by at least one, so the total number of `while` steps over the whole string is at most the total number of increases: at most m.

13.8.3 Searching with the prefix function

The **Knuth–Morris–Pratt** (KMP) search [15] runs the same loop over the haystack. It keeps k, the length of the longest prefix of the needle that ends at the current haystack position. Each new haystack character either extends that match, or makes the search fall back along the needle's borders exactly as in Equation (13.8); it never moves backwards in the haystack. When k reaches

m, the needle ends here, at position i, so it starts at $i - m + 1$; the search then falls back to $\pi[m - 1]$ to find overlapping occurrences. The whole search costs $O(n + m)$ and reads the haystack exactly once, from left to right. That is why **Listing 13.11** can read the haystack one character at a time from the input buffer and never store it: the needle and its prefix function are all it keeps.

Listing 13.11 `nhay.cpp`: KMP that streams the haystack character by character, storing only the needle and its prefix function.

```
// NHAY - A needle in the haystack: Knuth-Morris-Pratt, reading the haystack
// one character at a time so that it never has to be stored.
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

static char buf[1 << 16];
static size_t bufLen = 0, bufPos = 0;

int readChar() {
    if (bufPos == bufLen) {
        bufLen = fread(buf, 1, sizeof(buf), stdin);
        bufPos = 0;
        if (bufLen == 0) return -1;
    }
    return buf[bufPos++];
}

bool isSpace(int c) { return c == ' ' || c == '\n' || c == '\r' || c == '\t'; }

// pi[i] = length of the longest proper border of pattern[0..i].
vector<int> prefixFunction(const string& s) {
    vector<int> pi(s.size(), 0);
    for (size_t i = 1; i < s.size(); i++) {
        int k = pi[i - 1];
        while (k > 0 && s[i] != s[k]) k = pi[k - 1];  // try a shorter border
        if (s[i] == s[k]) k++;
        pi[i] = k;
    }
    return pi;
}

int main() {
    while (true) {
        int c = readChar();
        while (c != -1 && (c < '0' || c > '9')) c = readChar();
        if (c == -1) break;
        long long len = 0;
        for (; c >= '0' && c <= '9'; c = readChar()) len = len * 10 + (c - '0');
        string needle;
        c = readChar();
        while (c != -1 && isSpace(c)) c = readChar();
        for (; c != -1 && !isSpace(c); c = readChar()) needle += (char)c;
        vector<int> pi = prefixFunction(needle);
        int m = (int)needle.size(), k = 0;
```

Listing 13.11 (Continued) *nhay.cpp: KMP that streams the haystack character by character, storing only the needle and its prefix function.*

```
        c = readChar();
        while (c != -1 && isSpace(c)) c = readChar();
        for (long long i = 0; c != -1 && !isSpace(c); c = readChar(), i++) {
            while (k > 0 && (k == m || c != needle[k])) k = pi[k - 1];
            if (c == needle[k]) k++;
            if (k == m) printf("%lld\n", i - m + 1);  // a match ends at position i
        }
        putchar('\n');  // an empty line after each test case
        (void)len;      // the length line only announces the needle's size
    }
    return 0;
}
```

Listing 13.12 `nhay.py`: the same algorithm in Python, which reads each haystack as one string.

```
# NHAY - A needle in the haystack: Knuth-Morris-Pratt.
import sys


def prefix_function(s):
    """pi[i] = length of the longest proper border of s[0..i]."""
    pi = [0] * len(s)
    for i in range(1, len(s)):
        k = pi[i - 1]
        while k > 0 and s[i] != s[k]:
            k = pi[k - 1]  # fall back to a shorter border
        if s[i] == s[k]:
            k += 1
        pi[i] = k
    return pi


def kmp_positions(needle, hay):
    pi, m, k, found = prefix_function(needle), len(needle), 0, []
    for i, ch in enumerate(hay):
        while k > 0 and (k == m or ch != needle[k]):
            k = pi[k - 1]
        if ch == needle[k]:
            k += 1
        if k == m:
            found.append(i - m + 1)  # a match ends at position i
    return found


def main():
    data = sys.stdin.buffer.read().split()
    out = []
    for i in range(0, len(data) - 2, 3):
        needle, hay = data[i + 1], data[i + 2]
        out.extend(map(str, kmp_positions(needle, hay)))
        out.append("")  # an empty line after each test case
    sys.stdout.write("\n".join(out) + ("\n" if out else ""))


main()
```

The Python version, **Listing 13.12**, does store each haystack, because reading a character at a time in Python is slow; on a very large haystack it uses memory in proportion to its length. The byte strings from `sys.stdin.buffer` are indexed as integers, and the comparisons work unchanged.

13.8.4 Running it

```
2
na
bananana
3
xyz
abc
4
aaaa
aaaaaa
```

```
2
4
6


0
1
2
```

`na` occurs in `bananana` at positions 2, 4 and 6; `xyz` does not occur in `abc`, so that case prints only its empty line; and `aaaa` occurs in `aaaaaa` at 0, 1 and 2, three overlapping occurrences that a search which skips past each match would miss.

13.9 Repeated Prefixes: the Period of a String (PERIOD)

13.9.1 The problem in our own words

SPOJ problem PERIOD, "Period" [16], gives several strings of length up to 10^6. For every prefix of length $i \geq 2$ that is a repetition of a shorter block, that is, equal to some block written $K > 1$ times in a row, the program prints i and the largest such K, in increasing order of i. Each test starts with a line `Test case #k`.

13.9.2 The idea

The prefix function answers this for all prefixes at once. For a prefix of length L with longest border of length $\pi[L-1]$, the number

$$p = L - \pi[L-1] \tag{13.9}$$

is the length of its shortest *period*: shifting the prefix by p positions lines it up with itself, because the prefix and suffix of length $L-p$ coincide. The prefix is a repetition of a block exactly when p divides L (and $p < L$, that is, the border is not empty). In that case the block has length p and is repeated $K = L/p$ times, and no larger K is possible, because a shorter block would give a longer border. So the whole problem is: compute π, and for each L from 2 to n test whether $\pi[L-1] > 0$ and $L \bmod (L - \pi[L-1]) = 0$. Linear time, and the strings of 10^6 characters are no problem; **Listing 13.13** and **Listing 13.14** do exactly that.

Listing 13.13 `period.cpp`: the prefix function and Equation (13.9) for every prefix.

```cpp
// PERIOD - Every prefix that is a repetition: the prefix function answers all.
#include <cstdio>
#include <vector>
using namespace std;

int main() {
    int t;
    if (scanf("%d", &t) != 1) return 0;
    static char s[1000005];
    for (int tc = 1; tc <= t; tc++) {
        int n;
        if (scanf("%d %1000000s", &n, s) != 2) return 0;
        vector<int> pi(n, 0);
        for (int i = 1; i < n; i++) {
            int k = pi[i - 1];
            while (k > 0 && s[i] != s[k]) k = pi[k - 1];
            if (s[i] == s[k]) k++;
            pi[i] = k;
        }
        printf("Test case #%d\n", tc);
        for (int len = 2; len <= n; len++) {
            int p = len - pi[len - 1];  // the shortest period of this prefix
            if (pi[len - 1] > 0 && len % p == 0) printf("%d %d\n", len, len / p);
        }
        putchar('\n');
    }
    return 0;
}
```

Listing 13.14 `period.py`: the same in Python.

```python
# PERIOD - Every prefix that is a repetition: the prefix function answers all.
import sys


def main():
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    out = []
    for tc in range(1, t + 1):
        n, s = int(data[2 * tc - 1]), data[2 * tc]
        pi = [0] * n
        for i in range(1, n):
            k = pi[i - 1]
            while k > 0 and s[i] != s[k]:
                k = pi[k - 1]
            if s[i] == s[k]:
                k += 1
            pi[i] = k
        out.append("Test case #%d" % tc)
        for length in range(2, n + 1):
            p = length - pi[length - 1]  # the shortest period of this prefix
            if pi[length - 1] > 0 and length % p == 0:
                out.append("%d %d" % (length, length // p))
        out.append("")
    sys.stdout.write("\n".join(out) + "\n")


main()
```

13.9.3 Running it

```
2
6
ababab
7
aaaaaab
```

```
Test case #1
4 2
6 3

Test case #2
2 2
3 3
4 4
5 5
6 6
```

For `ababab`, the prefix `abab` is `ab` twice and `ababab` is `ab` three times; `aba` and `ababa` have period 2, which does not divide 3 or 5. For `aaaaaab` every prefix of `a`s is `a` repeated, and the last `b` breaks the pattern, so length 7 is not listed.

13.10 Counting Distinct Substrings: the Suffix Array (DISUBSTR)

13.10.1 The problem in our own words

SPOJ problem DISUBSTR, "Distinct Substrings" [17], gives up to 20 strings of length at most 1,000 and asks, for each, how many different non-empty substrings it has. A substring is a contiguous piece, and two occurrences of the same piece count once.

13.10.2 The idea: every substring is a prefix of a suffix

Every substring of s is a prefix of exactly one kind of suffix: the substring starting at position i is a prefix of the suffix $s[i..n-1]$. There are $n(n+1)/2$ pairs (start, length) in total, and the question is how many of them repeat an earlier one. Sort all n suffixes alphabetically; that sorted list of starting positions is the **suffix array** [1], [18]. Two suffixes that share a long common prefix end up next to each other in sorted order, and the prefixes a suffix shares with *any* earlier suffix are exactly the prefixes it shares with its neighbor just before it. If $\mathrm{lcp}[i]$ is the length of the longest common prefix of the i-th and $(i-1)$-th suffix in sorted order, then the i-th suffix contributes $\mathrm{lcp}[i]$ prefixes that were already counted, and

$$\#\,\text{distinct substrings} = \frac{n(n+1)}{2} - \sum_{i=1}^{n-1} \mathrm{lcp}\,[i]. \tag{13.10}$$

13.10.3 Building the arrays

Listing 13.15 builds the suffix array by *prefix doubling*: first rank suffixes by their first character, then by their first 2, 4, 8, … characters, each time sorting by the pair (rank of the first half, rank of the second half). After $\lceil \log_2 n \rceil$ rounds all ranks are distinct. With `std::sort` in each round the cost is $O(n \log^2 n)$, a few hundred thousand comparisons for $n = 1000$. The LCP array then comes from **Kasai's algorithm** [19] in $O(n)$: process the suffixes in order of their starting position, and use the fact that when the suffix starting at i shares h characters with its sorted predecessor, the suffix starting at $i+1$ shares at least $h-1$ with its own predecessor.

Listing 13.15 `disubstr.cpp`: suffix array by prefix doubling, LCP by Kasai, and Equation (13.10).

```
// DISUBSTR - Distinct substrings: suffix array (prefix doubling) plus LCP.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

vector<int> suffixArray(const string& s) {
    int n = (int)s.size();
    vector<int> sa(n), rank(n), tmp(n);
    for (int i = 0; i < n; i++) {
        sa[i] = i;
        rank[i] = (unsigned char)s[i];
    }
    for (int k = 1;; k <<= 1) {  // sort by the first 2k characters
        auto key = [&](int i) {
            return make_pair(rank[i], i + k < n ? rank[i + k] : -1);
        };
        sort(sa.begin(), sa.end(), [&](int a, int b) { return key(a) < key(b); });
        tmp[sa[0]] = 0;
        for (int i = 1; i < n; i++)
            tmp[sa[i]] = tmp[sa[i - 1]] + (key(sa[i - 1]) < key(sa[i]) ? 1 : 0);
        rank = tmp;
        if (rank[sa[n - 1]] == n - 1) break;  // all ranks distinct: done
    }
    return sa;
}

// Kasai's algorithm: lcp[i] = common prefix of suffixes sa[i-1] and sa[i].
vector<int> lcpArray(const string& s, const vector<int>& sa) {
    int n = (int)s.size();
    vector<int> rank(n), lcp(n, 0);
    for (int i = 0; i < n; i++) rank[sa[i]] = i;
    for (int i = 0, h = 0; i < n; i++) {
        if (rank[i] == 0) {
            h = 0;
            continue;
        }
        int j = sa[rank[i] - 1];
        while (i + h < n && j + h < n && s[i + h] == s[j + h]) h++;
        lcp[rank[i]] = h;
        if (h > 0) h--;  // the next suffix keeps at least h - 1 characters
    }
    return lcp;
}
```

Listing 13.15 (Continued) *disubstr.cpp: suffix array by prefix doubling, LCP by Kasai, and Equation (13.10).*

```
int main() {
    int t;
    string s;
    if (!(cin >> t)) return 0;
    getline(cin, s);  // the rest of the first line
    while (t-- && getline(cin, s)) {
        if (!s.empty() && s.back() == '\r') s.pop_back();
        long long n = (long long)s.size();
        if (n == 0) {
            cout << 0 << '\n';
            continue;
        }
        vector<int> sa = suffixArray(s), lcp = lcpArray(s, sa);
        long long total = n * (n + 1) / 2;  // all substrings, repeats included
        for (int x : lcp) total -= x;       // shared with the previous suffix
        cout << total << '\n';
    }
    return 0;
}
```

For $n \le 1000$ Python can afford a simpler construction, **Listing 13.16**: `sorted(range(n), key=lambda i: s[i:])` sorts the suffixes as strings, which costs $O(n^2 \log n)$ character comparisons in the worst case but runs in C inside the interpreter, and the LCP of neighbors is measured directly. The limit decides the method, as always: for $n = 10^5$ (SPOJ problem SUBST1 [20]) only the $O(n \log^2 n)$ construction and Kasai would do.

Listing 13.16 `disubstr.py`: sorted suffixes and neighbor LCPs, enough for $n \le 1000$.

```
# DISUBSTR - Distinct substrings: sorted suffixes plus longest common prefixes.
import sys


def distinct_substrings(s):
    n = len(s)
    order = sorted(range(n), key=lambda i: s[i:])  # fine for n <= 1000
    total = n * (n + 1) // 2  # every substring, repeats included
    for a, b in zip(order, order[1:]):
        h = 0
        while a + h < n and b + h < n and s[a + h] == s[b + h]:
            h += 1
        total -= h  # prefixes already seen in the suffix before
    return total


def main():
    lines = sys.stdin.read().split("\n")
    t = int(lines[0])
    out = [distinct_substrings(lines[i].rstrip("\r")) for i in range(1, t + 1)]
    print("\n".join(map(str, out)))


main()
```

Both programs read the strings *line by line* and use the characters as they are, because the statement allows characters other than letters; reading with `cin >> s` or `split()` would break a string that contains a space.

13.10.4 Running it

```
3
CACAO
AAAA
sate
```

```
12
4
10
```

For `CACAO`: 3 distinct letters, 3 distinct pairs (`CA` occurs twice), 3 triples, 2 quadruples and the whole string, 12 in total. `AAAA` has only 4 (one per length), and `sate`, with no repeated letter, has all $4 \cdot 5/2 = 10$.

13.11 Testing the Toolkit

The stress tests of `Chapter13/tests` follow the same plan as Chapter 12, with brute forces that trust nothing clever: `a ** b % 10` computed with the whole power for LASTDIG, counting with `gcd` for ETF, trial division (and, for numbers above 10^{12}, a list of known answers plus a many-base Fermat check) for PON, all pairs for CLOPPAIR, slicing at every position for NHAY, trying every block length for PERIOD, and a Python `set` of all substrings for DISUBSTR. The generators include the hard cases on purpose: Carmichael numbers and strong pseudoprimes for PON, points on a tiny 7×7 grid for CLOPPAIR (with the unique-closest-pair promise enforced), needles over a two-letter alphabet so that overlapping matches are frequent, and strings with digits and symbols for DISUBSTR. The results of the runs, 200 random inputs per problem with sanitizers enabled:

```
cloppair   200 random tests: C++ and Python agree with brute force
disubstr   200 random tests: C++ and Python agree with brute force
etf        200 random tests: C++ and Python agree with brute force
lastdig    200 random tests: C++ and Python agree with brute force
nhay       200 random tests: C++ and Python agree with brute force
pon        200 random tests: C++ and Python agree with brute force
period     200 random tests: C++ and Python agree with brute force
```

And one maximum-size input per problem, timed:

```
cloppair  max input  721.5 KB   C++  0.051 s   Python  1.065 s
disubstr  max input   19.6 KB   C++  0.020 s   Python  0.075 s
etf       max input  134.5 KB   C++  0.043 s   Python  0.164 s
lastdig   max input    0.4 KB   C++  0.002 s   Python  0.030 s
nhay      max input  977.5 KB   C++  0.019 s   Python  0.283 s
period    max input  976.6 KB   C++  0.091 s   Python  0.717 s
pon       max input    8.6 KB   C++  0.004 s   Python  0.049 s
```

Every C++ program runs in a tenth of a second or less on its largest input. Python is fast enough for LASTDIG, PON, ETF (thanks to the factorization approach), NHAY and DISUBSTR on our machine, and around one second for PERIOD and CLOPPAIR, where a judge with a strict limit may reject it. The NHAY timing uses a haystack of a million `a`s and a needle `aaa…ab`, the input that makes the naive search slowest: KMP does not care.

13.12 Using It for Real

Area and fence of a plot. Measure the corner posts (or trees) with a phone GPS, convert each position to meters east and north of one corner, and feed the corners in order to the shoelace formula for the area, and all trees to the convex hull for the fence; the fence length is the sum of

the hull's edge lengths. Round the coordinates to whole meters or decimeters and multiply by 10 to stay in integers.

Check digits and remainders. Many identification numbers end in a check digit computed as a weighted sum modulo 10 or 11; a mistyped digit changes the remainder. The same Equation (13.1) that keeps LASTDIG small lets a program verify such numbers without big integers.

Searching logs and chats. The KMP routine is a streaming filter: it reads text once and reports matches immediately, which makes it suitable for a log file that is still growing. For a single search in a stored string, the library function (`std::string::find`, Python's `str.find`) is usually fast enough in practice; knowing KMP matters when the judge's input is adversarial, when the text cannot be stored, or when the prefix function itself is the answer, as in PERIOD.

Recognizing the toolkit in new problems. "Print the answer modulo $10^9 + 7$" means modular arithmetic and fast exponentiation (and often the modular inverse $a^{p-2} \bmod p$, which is Equation (13.4) rearranged). "Up to 10^6, many queries" means a sieve. "Up to 10^{18}, is it prime" means Miller–Rabin. "Is this point left of that line", "area", "fence" mean the cross product. "Occurrences", "borders", "periods" mean the prefix function; "distinct substrings" and "longest repeated substring" mean the suffix array with LCP.

13.13 Common Mistakes, and How to Avoid Them

Overflow before the modulus. `a * b % m` with `int` operands overflows before `%` is applied; cast to `long long` first, and for moduli above about 3×10^9 use a 128-bit product.

Forgetting $b = 0$ or $a = 0$. Powers with exponent 0 are 1; zero to a positive power is 0. A loop that starts from the base rather than from 1 is wrong on the first case.

Negative remainders. In C++, `-7 % 3` is `-1`, not 2. When a subtraction can make a value negative, add the modulus before taking the remainder: `((x % m) + m) % m`. Python's `%` always returns a non-negative result for a positive modulus, which hides the bug when you port Python code to C++.

Fermat instead of Miller–Rabin. A Fermat test passes Carmichael numbers such as 561; always use Miller–Rabin with the deterministic bases.

Floating point where integers suffice. Comparing slopes with `double` or computing angles with `atan2` invites rounding errors on collinear points. Use the cross product on integers, and take a square root only for printing.

Wrong rounding in the output. Print exactly the number of decimals the statement asks for, with `printf("%.6f")` or `cout << fixed << setprecision(6)`.

Collinear points on the hull. Decide from the statement whether points on hull edges must be kept or dropped, and choose `<= 0` or `< 0` in the turn test accordingly.

Quadratic string code. Building a string with repeated `s = s + c` in a loop, calling `substr` inside a loop, or checking every position with `compare` can each be $O(n^2)$. Use the prefix function, and build strings with `+=` or `append`.

Reading a string that may contain spaces. Use `getline` (C++) or split the input into lines (Python) when the statement does not promise a single word.

13.14 Chapter Summary

- Modular arithmetic keeps numbers small, Equation (13.1); **fast exponentiation** computes $a^b \bmod m$ in $O(\log b)$ multiplications, Equation (13.2) (LASTDIG).
- **Euler's totient** follows from the prime factors, Equation (13.3); a sieve tabulates it for all $n \leq 10^6$ in $O(N \log\log N)$, and factorization with small primes answers single queries (ETF).
- **Miller–Rabin** with the first twelve primes as bases is a deterministic primality test for every 64-bit number; C++ needs a 128-bit product (PON).
- The **cross product** answers orientation questions exactly; with it come the **monotone-chain convex hull** in $O(n \log n)$ and the **shoelace** area, Equation (13.7).
- The **closest pair** of points is found by divide and conquer in $O(n \log n)$, using squared integer distances and a strip argument (CLOPPAIR).
- The **prefix function** gives linear-time pattern matching (**KMP**, NHAY) and the period of every prefix, Equation (13.9) (PERIOD).
- A **suffix array** with the **LCP** array counts distinct substrings by Equation (13.10) (DISUBSTR).
- All programs were stress-tested and timed; C++ is comfortable everywhere, and Python passes most of these problems on our machine.

13.15 Exercises

1. Compute the last digit of $13^{1000003}$ by hand, using the cycle of last digits.
2. Trace fast exponentiation for $3^{13} \bmod 7$: list the bits of 13, the running square and the result after each step.
3. Compute $\varphi(100)$ and $\varphi(97)$ with Equation (13.3), and check $\varphi(100)$ by counting the numbers in $1..100$ divisible by neither 2 nor 5.
4. Show that 561 is a Carmichael number by checking that $a^{560} \equiv 1 \pmod 3$, $\pmod{11}$ and $\pmod{17}$ for every a coprime to 561. (Hint: 560 is divisible by 2, 10 and 16.)
5. Run the Miller–Rabin test with base $a = 2$ on $n = 561$: write $560 = d \cdot 2^s$, compute the sequence Equation (13.5), and show that 2 is a witness.
6. Use Equation (13.6) to decide whether the segments from (0,0) to (4,4) and from (0,4) to (4,0) cross, and whether the segments from (0,0) to (2,2) and from (3,3) to (5,0) cross.
7. Compute with the shoelace formula the area of the quadrilateral (0,0), (5,0), (6,4), (1,3).
8. Compute the prefix function of `abacaba` and use it to list all positions where `aba` occurs in `abacabacaba`.
9. For the string `abcabcabcab`, list the prefixes that PERIOD reports and their K.
10. Count the distinct substrings of `banana` with Equation (13.10): write the six suffixes in sorted order, the LCP of each with its predecessor, and the result.

13.16 Solutions to the Exercises

1. Only the last digit of the base matters, so this is the last digit of $3^{1000003}$. The powers of 3 end in 3,9,7,1 and repeat every 4. $1{,}000{,}003 = 4 \cdot 250{,}000 + 3$, so the last digit is the third in the cycle: 7.

2. $13 = 1101_2$. Start with result $=1$, square $=3$. Bit 0 is 1: result $=1 \cdot 3 = 3$; square $=9 \bmod 7 = 2$. Bit 1 is 0: result stays 3; square $=4$. Bit 2 is 1: result $=3 \cdot 4 = 12 \bmod 7 = 5$; square $=16 \bmod 7 = 2$. Bit 3 is 1: result $=5 \cdot 2 = 10 \bmod 7 = 3$. So $3^{13} \bmod 7 = 3$. (Check: $3^6 \equiv 1$ by Fermat, so $3^{13} = 3^{12} \cdot 3 \equiv 3$.)

3. $100 = 2^2 \cdot 5^2$: $\varphi(100) = 100 \cdot \frac{1}{2} \cdot \frac{4}{5} = 40$. Counting: 50 numbers are even, 20 are multiples of 5, 10 are multiples of 10 and were counted twice, so $50 + 20 - 10 = 60$ share a factor with 100, and $100 - 60 = 40$ do not. 97 is prime, so $\varphi(97) = 96$.

4. For a coprime to 561, a is coprime to 3, 11 and 17. By Fermat, $a^2 \equiv 1 \pmod 3$, $a^{10} \equiv 1 \pmod{11}$, $a^{16} \equiv 1 \pmod{17}$. Since 560 is a multiple of 2, 10 and 16, $a^{560} \equiv 1$ modulo each of the three primes, hence modulo their product 561.

5. $560 = 35 \cdot 2^4$, so $d = 35$, $s = 4$. $2^{35} \bmod 561 = 263$; squaring: $263^2 \bmod 561 = 166$; $166^2 \bmod 561 = 67$; $67^2 \bmod 561 = 1$, and the last square is again 1. The sequence is 263,166,67,1,1: the first 1 is preceded by 67, which is neither 1 nor 560. So 67 is a square root of 1 modulo 561 other than ± 1, which is impossible modulo a prime: 2 is a witness, and 561 is composite. (Indeed $67^2 - 1 = 66 \cdot 68$, and $\gcd(66,561) = 33$ reveals factors.)

6. First pair: for the segment $A = (0,0)$, $B = (4,4)$, $\text{cross}(A, B, (0,4)) = 4 \cdot 4 - 4 \cdot 0 = 16 > 0$ and $\text{cross}(A, B, (4,0)) = 4 \cdot 0 - 4 \cdot 4 = -16 < 0$: the other segment's ends are on different sides. Symmetrically, for $C = (0,4)$, $D = (4,0)$: $\text{cross}(C, D, A) = 4 \cdot (-4) - (-4) \cdot 0 = -16$ and $\text{cross}(C, D, B) = 4 \cdot 0 - (-4) \cdot 4 = 16$. Both tests change sign, so the segments cross (at (2,2)). Second pair: $\text{cross}((0,0), (2,2), (3,3)) = 0$ and $\text{cross}((0,0), (2,2), (5,0)) = 2 \cdot 0 - 2 \cdot 5 = -10$; one end is on the line but beyond the segment's end (2,2), and the ends (0,0) and (2,2) relative to the second segment give $\text{cross}((3,3), (5,0), (0,0)) = 2 \cdot (-3) - (-3)(-3) = -15$ and $\text{cross}((3,3), (5,0), (2,2)) = 2 \cdot (-1) - (-3)(-1) = -5$, the same sign: no crossing.

7. Terms $x_i y_{i+1} - y_i x_{i+1}$: $(0 \cdot 0 - 0 \cdot 5) = 0$; $(5 \cdot 4 - 0 \cdot 6) = 20$; $(6 \cdot 3 - 4 \cdot 1) = 14$; $(1 \cdot 0 - 3 \cdot 0) = 0$. Sum 34, area $34/2 = 17$.

8. $\pi(\text{abacaba}) = [0,0,1,0,1,2,3]$. For the search, the prefix function of the needle `aba` is [0,0,1]; scanning `abacabacaba`, the match length reaches 3 at positions 2, 6 and 10, so `aba` starts at 0, 4 and 8.

9. The shortest period of every prefix of length at least 3 is 3 (block `abc`). It divides the length for $L = 6$ ($K = 2$) and $L = 9$ ($K = 3$); the prefixes of lengths 2, 3, 4, 5, 7, 8, 10, 11 are not repetitions ($L = 3$ has an empty border). Output: `6 2` and `9 3`.

10. Sorted suffixes: `a`, `ana`, `anana`, `banana`, `na`, `nana`. LCPs with the predecessor: `ana`/`a` 1, `anana`/`ana` 3, `banana`/`anana` 0, `na`/`banana` 0, `nana`/`na` 2; sum 6. $n(n+1)/2 = 21$, so there are $21 - 6 = 15$ distinct substrings.

References

[1] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

[2] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 2. Singapore: Lulu, 2020.

[3] SPOJ, "LASTDIG — The last digit," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/LASTDIG/

[4] SPOJ, "ETF — Euler Totient Function," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/ETF/

[5] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, 6th ed., revised by D. R. Heath-Brown and J. H. Silverman. Oxford, UK: Oxford University Press, 2008.

[6] SPOJ, "PON — Prime or Not," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/PON/

[7] M. O. Rabin, "Probabilistic algorithm for testing primality," *Journal of Number Theory*, vol. 12, no. 1, pp. 128–138, 1980.

[8] J. Sorenson and J. Webster, "Strong pseudoprimes to twelve prime bases," *Mathematics of Computation*, vol. 86, no. 304, pp. 985–1003, 2017.

[9] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Berlin, Germany: Springer, 2008.

[10] A. M. Andrew, "Another efficient algorithm for convex hulls in two dimensions," *Information Processing Letters*, vol. 9, no. 5, pp. 216–219, 1979.

[11] SPOJ, "CLOPPAIR — Closest Point Pair," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/CLOPPAIR/

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

[13] M. I. Shamos and D. Hoey, "Closest-point problems," in *Proc. 16th Annual Symposium on Foundations of Computer Science*, 1975, pp. 151–162.

[14] SPOJ, "NHAY — A Needle in the Haystack," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/NHAY/

[15] 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.

[16] SPOJ, "PERIOD — Period," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/PERIOD/

[17] SPOJ, "DISUBSTR — Distinct Substrings," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/DISUBSTR/

[18] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[19] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," in *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2001, pp. 181–192.

[20] SPOJ, "SUBST1 — New Distinct Substrings," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/SUBST1/

Chapter 14

Getting to Accepted: End-to-End Case Studies and Where to Go Next

The previous chapters took the craft apart: reading a statement, budgeting time from the constraints, fast input and output, compiler flags, memory, classic traps, stress testing, and the algorithmic toolkits of dynamic programming, greedy choice, binary search, graphs, numbers, geometry and strings. This last chapter puts the pieces back together. It follows three SPOJ problems from the first reading of the statement to a program we would submit with confidence, doing every step in the order and with the care that a contest demands, including the step most beginners skip: proving to ourselves, with a brute force and a stress test, that the program is right before the judge tells us. One of the three case studies has a bug planted in it on purpose, so that you can watch a stress test find it and shrink it to a two-line input. The second half of the chapter looks outward: where to practice after SPOJ, which contests run in 2026 and how to enter them from Indonesia, which books to read next, and how to organize the months of practice that turn the techniques of this book into habits. By the end of the chapter you will be able to run the whole workflow on a new problem on your own, and you will have a concrete plan for what to do next.

14.1 From a First Submission to a Habit

Dimas is a second-year informatics student in Surabaya. In his first month on SPOJ he submitted TEST, the problem that echoes numbers until it meets 42, and got Accepted on the third try, after learning that the judge compares output exactly. Then came a streak of Wrong Answers on problems he was sure he had solved, a Time Limit Exceeded on a problem that ran instantly on his laptop, and a Runtime Error that appeared only on the judge. Each verdict taught him something, and each lesson is a chapter of this book. What changed him from someone who collects verdicts into someone who expects Accepted was not a new algorithm. It was a routine: the same sequence of steps, done every time, on paper and in the terminal, before pressing Submit.

His teammates for the Indonesia National Contest (INC) in October have the same goal. They will have five hours for ten or more problems, one computer for three people, and no feedback except the verdicts. In that setting a routine is not a luxury. A wrong submission costs twenty penalty minutes in ICPC-style scoring; a bug found by a stress test on your own machine costs nothing. This chapter is that routine, written down and practiced three times.

14.2 The Workflow

Figure 14.1 shows the eight steps. The first three happen on paper, the next four at the keyboard, and the last is the only one that involves the judge. The red and amber arrows are the ways back: a mismatch in the stress test sends you back to the fast solution with a small failing input in hand, and a verdict other than Accepted sends you back to reading (for Wrong Answer or Runtime Error) or to the complexity budget (for Time or Memory Limit Exceeded).

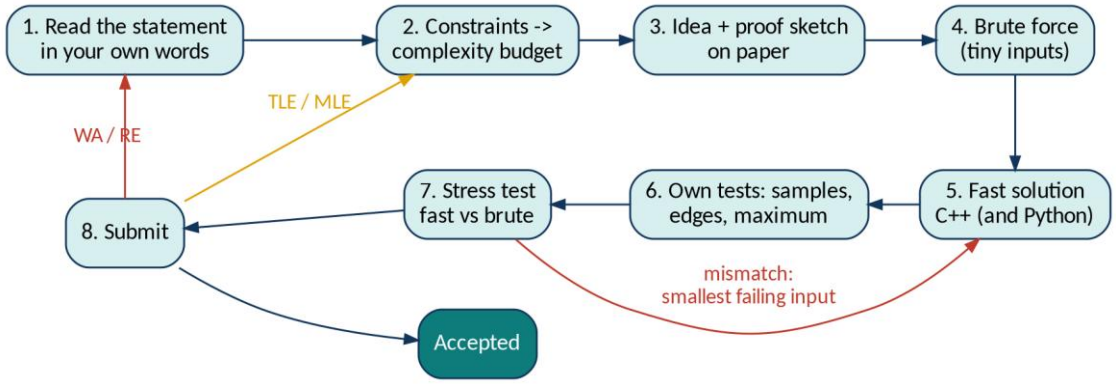


Figure 14.1 The end-to-end workflow used in every case study of this chapter, with the ways back after a failed stress test or a verdict other than Accepted.

Table 14.1 expands each step into the questions to answer and the chapter of this book that teaches it. It is meant to be copied onto a card and kept next to the keyboard until it is no longer needed.

Table 14.1 The eight steps of the workflow, the questions each one answers, and where this book teaches them.

Step	Questions to answer	Chapter
1. Read	What exactly is asked? What are input and output formats, including separators, case lines and termination? Restate it in two sentences.	2
2. Budget	Largest n, number of test cases, time and memory limits: which complexity fits? Which types (overflow)?	3, 6
3. Idea	Which pattern (DP, greedy, search, graph, number theory, strings)? Why is it correct? Hand-run the samples.	9–13
4. Brute force	What is the simplest program that is obviously correct on tiny inputs?	8
5. Fast solution	Clean C++ with fast I/O; Python if the budget allows.	4, 5
6. Own tests	Samples, smallest and largest cases, equal values, the traps you know.	7
7. Stress test	Hundreds of random small inputs: fast versus brute force, with sanitizers.	8
8. Submit	Right file, right language, no debug output, the output format one last time.	1, 4

The time spent on steps 1 to 4 feels wasted to a beginner who wants to type code. It is the opposite. A statement misread in step 1 costs the whole problem; a complexity misjudged in step 2 costs a rewrite; and a brute force written in step 4 takes five minutes and saves an hour of staring at a Wrong Answer. Laaksonen [1] and Halim et al. [2] give the same advice in almost the same words: think first, and test your own program before the judge does.

14.3 Case Study 1: Prime Path (PPATH)

14.3.1 Step 1: reading

SPOJ problem PPATH, “Prime Path” [3], is a puzzle about four-digit primes. We start from one four-digit prime and want to reach another by changing one digit at a time, and every number along the way must itself be a four-digit prime (so the first digit may never become 0). Each change costs one unit; the program prints the minimum cost, or the word `Impossible` if the target cannot be reached. The input starts with the number of test cases, at most 100, followed by one line per test with the two primes.

In two sentences of our own: *the four-digit primes are the vertices of a graph, and two primes are joined when they differ in exactly one digit; for each test, print the length of the shortest path between the two given primes, or* `Impossible`. That restatement already contains the algorithm, which is why step 1 is worth doing slowly.

14.3.2 Step 2: the budget

There are 1,061 four-digit primes. Each has at most $4 \times 9 = 36$ numbers that differ from it in one digit, so the graph has about a thousand vertices and at most $1{,}061 \times 36/2 \approx 19{,}000$ edges. A BFS costs $O(n + m)$, some tens of thousands of steps, and 100 tests cost a few million: nothing. Even Python fits easily. Types: all numbers are below 10^4, and the answer is at most the number of vertices, so `int` is plenty.

14.3.3 Step 3: the idea and why it is right

All edges cost one, so this is breadth-first search from Chapter 12, on a graph that is never stored: the neighbors of a number are generated on the fly by replacing each of its four digits with each other digit and keeping the results that are prime. Primality comes from a sieve of Eratosthenes below 10^4, computed once. BFS gives the minimum number of changes by the key insight of Section 12.4.2, and if the target is never reached the queue empties and the answer is `Impossible`. A start equal to the target gives 0 without any move.

By hand, with our own first test 1009 → 9001: the neighbors of 1009 that are prime are 1019, 1039, 1049, 1069, 1109, 1409, 1609, 1709, 5009 and 8009; none is the target. BFS finds a path of length 4,

$$1009 \to 1609 \to 1601 \to 9601 \to 9001, \tag{14.1}$$

and no shorter one exists. The two numbers differ in the thousands and the units digit, so at least two changes are needed; the only two-change routes pass through 9009 or 1001, and neither is prime ($9009 = 9 \cdot 1001$ and $1001 = 7 \cdot 11 \cdot 13$); and BFS checks exhaustively that no route of three changes exists either.

14.3.4 Step 4: the brute force

The brute force must be obviously right, not fast. Ours, in `Chapter14/tests/brute.py`, does not use BFS at all: it tests primality by trial division, sets the distance of the source to 0 and every other prime to "infinity", and then repeatedly relaxes every prime's neighbors, Bellman–Ford style, until no distance changes. It is correct by the same argument as Bellman–Ford [4], shares no code with the fast solution, and takes a noticeable fraction of a second per source, which is fine for a tester.

14.3.5 Step 5: the fast solution

Listing 14.1 computes the sieve once and runs one BFS per test. Changing digit number `pos` (1, 10, 100 or 1000) of `u` is arithmetic: subtract the old digit times its place value, add the new one. The loop over new digits starts at 1 for the thousands place, which is the "no leading zero" rule of the statement in one expression.

Listing 14.1 `ppath.cpp`: a sieve below 10^4 and one breadth-first search per test.

```cpp
// PPATH - Prime path: sieve the four-digit primes, then breadth-first search.
#include <cstdio>
#include <queue>
#include <vector>
using namespace std;

int main() {
    vector<bool> composite(10000, false);  // sieve of Eratosthenes below 10^4
    composite[0] = composite[1] = true;
    for (int p = 2; p * p < 10000; p++)
        if (!composite[p])
            for (int q = p * p; q < 10000; q += p) composite[q] = true;
    const int POW10[4] = {1, 10, 100, 1000};
    int t;
    if (scanf("%d", &t) != 1) return 0;
    while (t--) {
        int from, to;
        if (scanf("%d %d", &from, &to) != 2) return 0;
        vector<int> dist(10000, -1);
        queue<int> q;
        dist[from] = 0;
        q.push(from);
        while (!q.empty() && dist[to] == -1) {
            int u = q.front();
            q.pop();
            for (int pos = 0; pos < 4; pos++) {  // change one digit at a time
                int digit = u / POW10[pos] % 10;
                int base = u - digit * POW10[pos];
                for (int d = (pos == 3 ? 1 : 0); d <= 9; d++) {  // no leading zero
                    int v = base + d * POW10[pos];
                    if (composite[v] || dist[v] != -1) continue;
                    dist[v] = dist[u] + 1;
                    q.push(v);
                }
            }
        }
        if (dist[to] == -1)
            puts("Impossible");
        else
            printf("%d\n", dist[to]);
    }
    return 0;
}
```

Listing 14.2 `ppath.py`: the same search in Python, with a generator for the neighbors and a slice assignment in the sieve.

```
# PPATH - Prime path: sieve the four-digit primes, then breadth-first search.
import sys
from collections import deque

composite = bytearray(10000)  # sieve of Eratosthenes below 10^4
composite[0] = composite[1] = 1
for p in range(2, 100):
    if not composite[p]:
        composite[p * p::p] = b"\x01" * len(range(p * p, 10000, p))


def neighbors(u):
    for pos in (1, 10, 100, 1000):  # change one digit at a time
        base = u - (u // pos % 10) * pos
        for d in range(1 if pos == 1000 else 0, 10):  # no leading zero
            v = base + d * pos
            if not composite[v]:
                yield v


def cost(src, dst):
    dist = {src: 0}
    q = deque([src])
    while q and dst not in dist:
        u = q.popleft()
        for v in neighbors(u):
            if v not in dist:
                dist[v] = dist[u] + 1
                q.append(v)
    return dist.get(dst)


def main():
    data = sys.stdin.read().split()
    out = []
    for i in range(int(data[0])):
        c = cost(int(data[1 + 2 * i]), int(data[2 + 2 * i]))
        out.append("Impossible" if c is None else str(c))
    print("\n".join(out))


main()
```

The Python sieve of **Listing 14.2** uses slice assignment, `composite[p * p::p] = ...`, which marks all multiples of `p` in one call implemented in C; this is the idiomatic way to write a fast sieve in pure Python.

14.3.6 Steps 6 and 7: our own tests, then the stress test

Our sample has a long path, a zero-length path (source equals target) and a second long path:

```
3
1009 9001
2003 2003
4001 8837
```

```
4
0
4
```

The first answer matches the hand trace Equation (14.1). Is `Impossible` ever printed? Among four-digit primes the graph turns out to be connected, so a correct program never prints it; a program that forgets the word entirely would still be accepted, and one that prints `impossible` in

lower case would never be caught. That is not a reason to be careless: the statement says the word may be needed, and the code prints exactly the statement's spelling.

The stress test compared C++, Python and the brute force on 200 random inputs of up to five tests each, then timed one input with the maximum of 100 tests:

```
ppath      200 random tests: C++ and Python agree with brute force
ppath     max input    1.0 KB   C++  0.016 s   Python  0.185 s
```

14.3.7 Step 8: submitting

Before submitting, the last check: the program reads the number of tests first, prints one line per test, has no debugging output left on `stdout`, and the file submitted is `ppath.cpp` with the language set to C++ (GCC). If this had been Python, the timing above (under 0.2 seconds for 100 tests) would say it is safe too.

14.4 Case Study 2: The Longest ABC Path (ABCPATH)

14.4.1 Steps 1 to 3

SPOJ problem ABCPATH, "ABC Path" [5], gives grids of uppercase letters, each up to 50×50, one after another until a line `0 0`. From a cell containing `A`, a path may step to any of the eight neighboring cells (including diagonals) if the letter there is the next letter of the alphabet: `A` to `B`, `B` to `C`, and so on. For each grid the program prints `Case C: X`, where `C` counts the grids from 1 and `X` is the length of the longest such path, which is 0 when the grid has no `A`.

Budget: 2,500 cells, eight neighbors each. A plain search that tries every path could be slow on a grid where many paths share their tails, so we memoize: let $L(r, c)$ be the length of the longest path *starting* at cell (r, c). Then

$$\begin{aligned} L(r,c) \\ &= 1 + \max(\{0\} \\ \cup \{\, L(r',c') &: (r',c') \text{ adjacent to } (r,c),\ \text{letter}(r',c') = \text{letter}(r,c) + 1 \,\}) \end{aligned} \tag{14.2}$$

and the answer is the maximum of L over the `A` cells. The recursion always moves to a strictly larger letter, so it cannot loop and is at most 26 levels deep; each cell is computed once, so a grid costs $O(8 \cdot HW)$, twenty thousand steps. This is dynamic programming on a directed acyclic graph, exactly the pattern of Chapter 9: the letters give the order.

14.4.2 Steps 4 and 5

The brute force is Equation (14.2) without the memo: plain recursion that tries every path. It is exponential in the worst case, but on grids of at most 5×5 with five different letters it finishes instantly. The fast solution in C++ is the memoized recursion, **Listing 14.3**.

Listing 14.3 `abcpath.cpp`: memoized search; the memo table is cleared for every grid.

```cpp
// ABCPATH - Longest path of consecutive letters from 'A': memoized search.
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

int h, w;
char grid[52][52];
int best[52][52];  // longest path starting at (r, c); -1 = not computed yet

int longest(int r, int c) {
    int& res = best[r][c];
    if (res != -1) return res;
    res = 1;
    for (int dr = -1; dr <= 1; dr++)
        for (int dc = -1; dc <= 1; dc++) {  // eight neighbors (and (0,0), skipped)
            int nr = r + dr, nc = c + dc;
            if (nr < 0 || nr >= h || nc < 0 || nc >= w) continue;
            if (grid[nr][nc] == grid[r][c] + 1) res = max(res, 1 + longest(nr, nc));
        }
    return res;
}

int main() {
    for (int tc = 1; scanf("%d %d", &h, &w) == 2 && (h || w); tc++) {
        for (int r = 0; r < h; r++)
            if (scanf("%51s", grid[r]) != 1) return 0;
        memset(best, -1, sizeof(best));  // forget the previous grid
        int answer = 0;
        for (int r = 0; r < h; r++)
            for (int c = 0; c < w; c++)
                if (grid[r][c] == 'A') answer = max(answer, longest(r, c));
        printf("Case %d: %d\n", tc, answer);
    }
    return 0;
}
```

In Python, we avoid recursion altogether. Because every step goes from a letter to the next letter, processing the cells from `Z` down to `A` guarantees that when a cell is processed, all the cells it can step to are already finished. **Listing 14.4** sorts the cells by letter in decreasing order and fills the table in that order: the same recurrence, evaluated bottom-up.

Listing 14.4 `abcpath.py`: the same recurrence evaluated bottom-up, from `Z` down to `A`.

```
# ABCPATH - Longest path of consecutive letters from 'A': memoized search.
import sys


def solve(grid):
    h, w = len(grid), len(grid[0])
    best = [[0] * w for _ in range(h)]  # longest path starting at (r, c)
    order = sorted(((r, c) for r in range(h) for c in range(w)),
                   key=lambda rc: grid[rc[0]][rc[1]], reverse=True)
    for r, c in order:  # from 'Z' down to 'A': every successor is already done
        res = 1
        for nr in range(max(0, r - 1), min(h, r + 2)):
            for nc in range(max(0, c - 1), min(w, c + 2)):
                if ord(grid[nr][nc]) == ord(grid[r][c]) + 1:
                    res = max(res, 1 + best[nr][nc])
        best[r][c] = res
    return max([best[r][c] for r in range(h) for c in range(w)
                if grid[r][c] == "A"], default=0)


def main():
    data = sys.stdin.read().split()
    pos, tc, out = 0, 1, []
    while pos + 1 < len(data):
        h, w = int(data[pos]), int(data[pos + 1])
        pos += 2
        if h == 0 and w == 0:
            break
        grid = data[pos:pos + h]
        pos += h
        out.append("Case %d: %d" % (tc, solve(grid)))
        tc += 1
    print("\n".join(out))


main()
```

```
3 4
ABCE
DCDF
AEFG
1 2
BC
0 0
```

```
Case 1: 7
Case 2: 0
```

In the first grid the path `A B C D E F G` uses the top-left `A`, one of the `C`s, the `D` at the right of the middle row, the `E` in the top-right corner, then `F` and `G` down the right edge: 7 letters. The second grid has no `A`, so its answer is 0, a case worth having in the sample because a program that initializes its answer to 1 would fail it.

14.4.3 Step 7 at work: catching a planted bug

Now the instructive part. `abcpath_wrong.cpp` is the same program with one line moved: the memo table is cleared once at the start of `main` instead of once per grid. It is exactly the bug a tired contestant writes, and it passes the sample of the statement, which has a single grid. Our own sample above also passes, by luck. The stress test does not rely on luck:

```
MISMATCH in abcpath (C++), test 1
3 3
DAC
ACC
DBC
1 3
DDA
4 5
ADEED
DECDE
DEEAE
BBAEA
5 4
BDBA
DDEC
EDCE
EDDD
BCCA
0 0

expected: Case 1: 4 Case 2: 1 Case 3: 2 Case 4: 3
got:      Case 1: 4 Case 2: 1 Case 3: 2 Case 4: 2
```

The fourth grid of a random input got 2 instead of 3. Why? The cells of the new grid reuse memo entries computed for the *previous* grids at the same coordinates, and those values describe paths in a different grid. The stress test found the bug in its second random input. The failing input is still a little large to reason about, so the next step is to *shrink* it: remove grids and cells while the programs still disagree. A few minutes of trying (or a small loop, as Chapter 8 shows) gives a failing input of two lines of letters:

```
1 2
AB
1 1
A
0 0
```

The correct program and the buggy one on that input:

```
Case 1: 2
Case 2: 1
Case 1: 2
Case 2: 2
```

In the first grid, cell (0,0) holds `A` with a `B` next to it, so its memo entry becomes 2. The second grid is a single `A` at the same cell (0,0), and the buggy program reads the stale 2. Once the bug is this small, the fix is obvious: clear the memo for every grid, the line `memset(best, -1, sizeof(best));` inside the loop of **Listing 14.3**. This is the general lesson of Chapter 8 seen in practice: *every piece of state that belongs to one test case must be reset for the next*, and random inputs with several test cases per file are the cheapest way to check it.

14.5 Case Study 3: Ones and Zeros (ONEZERO)

14.5.1 Steps 1 and 2

SPOJ problem ONEZERO, “Ones and zeros” [6], gives about 1,000 numbers n with $1 \le n \le 20{,}000$. For each, the program must print the smallest positive multiple of n whose decimal digits are all 0 or 1. For $n = 2$ the answer is 10; for $n = 7$ it is 1001, because $1001 = 7 \times 143$.

The budget question is harder than it looks, because it concerns the *answer*, not the input. How long can the answer be? For $n = 9999$ it is 36 ones, a 36-digit number. No built-in integer type of

C++ holds that, so any approach that stores the candidate numbers themselves as integers is in danger.

14.5.2 Step 3: search over remainders

The key observation is that for divisibility by n only the remainder modulo n matters, and appending a digit d to a number x changes its remainder in a way that depends only on the old remainder:

$$(10x + d) \bmod n = (10\,(x \bmod n) + d) \bmod n. \tag{14.3}$$

So think of the n possible remainders as the vertices of a graph, with an edge "append 0" and an edge "append 1" out of each. The number 1 has remainder $1 \bmod n$, and we want to reach remainder 0 by the shortest sequence of appended digits, since fewer digits means a smaller number. That is breadth-first search again, on a graph of $n \le 20{,}000$ vertices, never on the huge numbers themselves. **Figure 14.2** shows the search for $n = 7$.

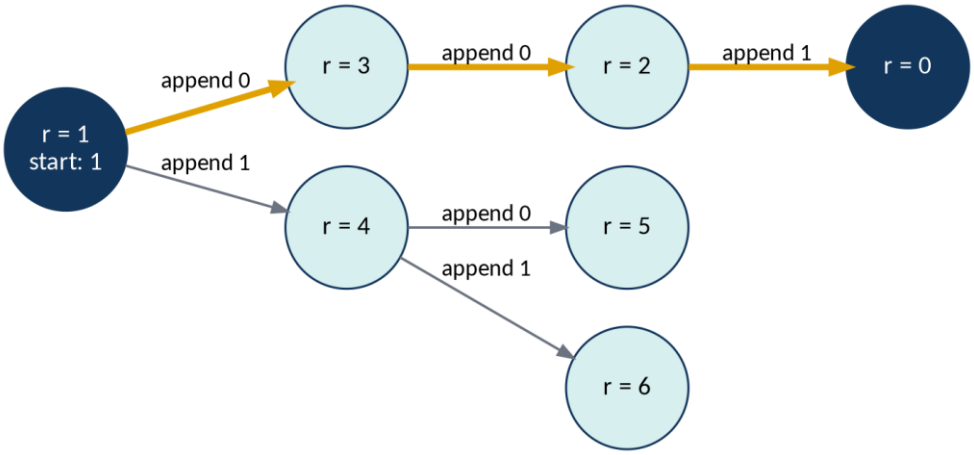


Figure 14.2 Breadth-first search over remainders modulo 7: the highlighted path appends 0, 0, 1 to the starting 1 and reaches remainder 0, so the answer is 1001.

Three facts make this correct.

1. *The search ends.* Among the $n + 1$ numbers 1,11,111, … (with up to $n + 1$ ones), two have the same remainder by the pigeonhole principle, and their difference is a number of the form 11 … 100 … 0, a multiple of n written with 0s and 1s. So an answer exists with at most n digits, and BFS, which visits each remainder at most once, finds one.
2. *BFS finds the shortest answer, and among equally short ones the smallest.* BFS reaches numbers in order of length; within one length, trying "append 0" before "append 1" visits them in increasing numeric order, because the queue keeps the order of the prefixes.
3. *Pruning by remainder is safe.* If two numbers $x < y$ have the same remainder, then for any digits appended to both, the results have equal remainders and the one built on x is smaller. So once a remainder has been reached, any later number with the same remainder can be forgotten.

To print the answer, each remainder stores its parent remainder and the digit that led to it; walking back from remainder 0 and reversing gives the digits.

14.5.3 Step 5, and the tempting wrong way

Before the correct program, the wrong one, because it is the one most people write first. **Listing 14.5** runs BFS over the candidate numbers 1,10,11,100, … stored as 64-bit unsigned integers.

Listing 14.5 `onezero_wrong.cpp`: BFS over the numbers themselves, which overflows silently once they pass twenty digits.

```cpp
// A tempting but wrong ONEZERO: breadth-first search over the numbers themselves.
#include <cstdio>
#include <queue>
using namespace std;

int main() {
    int k;
    if (scanf("%d", &k) != 1) return 0;
    while (k--) {
        int n;
        if (scanf("%d", &n) != 1) return 0;
        queue<unsigned long long> q;  // 1, 10, 11, 100, 101, 110, 111, ...
        q.push(1);
        while (true) {
            unsigned long long x = q.front();
            q.pop();
            if (x % n == 0) {
                printf("%llu\n", x);
                break;
            }
            q.push(x * 10);      // overflows silently beyond 20 digits
            q.push(x * 10 + 1);
        }
    }
    return 0;
}
```

It is correct for every n whose answer has at most 19 digits. For larger answers, `x * 10` wraps around modulo 2^{64} without any warning, the "numbers" in the queue are no longer made of 0s and 1s, and the program eventually prints some wrapped value that happens to be divisible by n. Worse, the queue doubles with every digit, so even without overflow a 36-digit answer would need 2^{36} queue entries. Here is the wrong program on our sample, next to the correct one:

```
5
1
2
7
12
9999
```

```
1
10
1001
11100
13506811700559478884
1
10
1001
11100
111111111111111111111111111111111111
```

The first four answers agree; for $n = 9999$ the wrong program prints a 20-digit number full of other digits. **Listing 14.6** is the correct program, following Section 14.5.2 line by line, and **Listing 14.7** is its Python twin.

Listing 14.6 `onezero.cpp`: BFS over the remainders modulo n with parent links, then the digits read back from remainder 0.

```cpp
// ONEZERO - Smallest multiple of n written with digits 0 and 1:
// breadth-first search over remainders modulo n, never over the huge numbers.
#include <algorithm>
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

string smallestZeroOne(int n) {
    vector<int> parent(n, -2);  // -2: remainder not reached yet, -1: the start
    vector<char> digit(n);
    vector<int> queue;
    queue.reserve(n);
    int start = 1 % n;
    parent[start] = -1;
    digit[start] = '1';
    queue.push_back(start);
    for (size_t head = 0; head < queue.size() && parent[0] == -2; head++) {
        int r = queue[head];
        for (int d = 0; d <= 1; d++) {  // try appending 0 before 1: smaller first
            int nr = (r * 10 + d) % n;
            if (parent[nr] != -2) continue;
            parent[nr] = r;
            digit[nr] = (char)('0' + d);
            queue.push_back(nr);
        }
    }
    string s;
    for (int r = 0; r != -1; r = parent[r]) s += digit[r];  // walk back to start
    reverse(s.begin(), s.end());
    return s;
}

int main() {
    int k;
    if (scanf("%d", &k) != 1) return 0;
    while (k--) {
        int n;
        if (scanf("%d", &n) != 1) return 0;
        puts(smallestZeroOne(n).c_str());
    }
    return 0;
}
```

Listing 14.7 `onezero.py`: the same search in Python, with a list and a head index as the queue.

```python
# ONEZERO - Smallest multiple of n written with digits 0 and 1:
# breadth-first search over remainders modulo n, never over the huge numbers.
import sys


def smallest_zero_one(n):
    parent = [-2] * n  # -2: remainder not reached yet, -1: the start
    digit = [""] * n
    start = 1 % n
    parent[start], digit[start] = -1, "1"
    queue = [start]
    head = 0
    while head < len(queue) and parent[0] == -2:
        r = queue[head]
        head += 1
        for d in (0, 1):  # try appending 0 before 1: smaller numbers first
            nr = (r * 10 + d) % n
            if parent[nr] == -2:
                parent[nr], digit[nr] = r, str(d)
                queue.append(nr)
    s, r = [], 0
    while r != -1:  # walk back from remainder 0 to the start
        s.append(digit[r])
        r = parent[r]
    return "".join(reversed(s))


def main():
    data = sys.stdin.read().split()
    print("\n".join(smallest_zero_one(int(x)) for x in data[1:1 + int(data[0])]))


main()
```

14.5.4 Steps 6 and 7: a checker instead of a brute force

For small n a brute force exists: count $k = 1,2,3,\ldots$ and read the binary digits of k as a decimal number, which lists all 0/1 numbers in increasing order; the first multiple of n is the answer. It agrees with both programs on 200 random tests with $n \le 300$. But the interesting inputs are the large ones, where the brute force would count to 2^{36}. For them we write a **checker** instead: a program that does not compute the answer, but verifies a given answer. `check_onezero.py` checks, for every n from 1 to 20,000, that the printed answer contains only 0s and 1s, starts with 1, is divisible by n (Python integers have no size limit), and that no 0/1 number with fewer digits is divisible by n, which it establishes by tracking the set of remainders reachable with each number of digits. The last check proves minimality in length; the BFS order argument of Section 14.5.2 covers the rest. On all 20,000 values:

```
checked 20000 values of n, 0 wrong
checked 20000 values of n, 0 wrong
n = 792: wrong answer 420545668853801304
n = 999: wrong answer 4976921130795323847
n = 1188: wrong answer 8877380742563353020
checked 20000 values of n, 162 wrong
```

The first line checks the Python program, the second the C++ program, and the last four lines the tempting wrong program, which gives a wrong answer for 162 of the 20,000 values of n: exactly the 162 values whose answers have more than 20 digits, the length at which `x * 10` first leaves the range of a 64-bit unsigned integer. Checking all 20,000 values takes about half a minute and covers every input the judge can possibly give; for a problem with a small input space, exhausting it is the strongest test there is.

The time on a maximum input of 1,000 queries (the last line of the stress run) is the last piece of information:

```
onezero   max input   5.3 KB   C++  0.125 s   Python  2.477 s
```

C++ is instant; Python needs about 2.5 seconds for 1,000 queries of the worst size on our machine, because each query runs a BFS over up to 20,000 remainders in the interpreter. If the judge's limit is tight, a precomputation helps: many queries repeat, and a `dict` of answers already computed makes each repeated query free.

14.6 When the Verdict Is Not Accepted

Even with the workflow, some submissions fail. The verdict is information, and **Table 14.2** is a map from each verdict to its most likely causes and the next thing to do, collecting the advice of the earlier chapters in one place.

Table 14.2 What each verdict usually means, what to do next, and where this book discusses it.

Verdict	Most likely causes	Next action	Chapter
Wrong Answer	Misread format; overflow; missing reset between tests; unhandled edge case ($n = 1$, empty, equal values); wrong rounding	Reread the statement; list edge cases; stress test with several test cases per file	2, 7, 8
Time Limit Exceeded	Algorithm too slow for the real maximum; slow I/O; hidden quadratic step	Recompute the budget; time a maximum input locally; fast I/O	3, 4
Runtime Error	Out-of-bounds index; recursion too deep; division by zero; reading past end of input	Sanitizers; explicit stack; check array sizes against constraints	5, 7
Memory Limit Exceeded	Adjacency matrix for large n; `long long` where `int` suffices; storing what can be streamed	Recompute memory in bytes; stream the input	6
Compilation Error	Compiler extensions; missing headers on the judge's compiler; wrong language selected	Compile with the judge's flags locally	5

One rule is worth stating separately: never resubmit the same program hoping for a different verdict, and never change two things at once. Change one thing, retest locally, and submit again.

14.7 Where to Go Next

14.7.1 Judges for daily practice

SPOJ remains a good archive of classical problems, and everything in this book transfers directly to the other judges. Each has a character of its own, summarized in **Table 14.3**.

- **Codeforces** [7] runs short rated contests several times a month, divided by level (Div. 4 for beginners up to Div. 1 for the strongest), with a rating that follows your progress, a huge problem archive tagged by topic and difficulty, and editorials after each round. It is the busiest judge in the world and the natural next step after this book.
- **AtCoder** [8] holds the AtCoder Beginner Contest (ABC) almost every weekend: 100 minutes, typically seven problems A to G, rated for participants below 2000 (for example ABC 473 on August 29, 2026 [9]). Its problems are clean, carefully tested and well suited to steady practice. In 2026 AtCoder reorganized its Regular Contest into three levels, ARC++ (rated 1600–2999), ARC (1200–2799) and ARC– (800–2399), starting with ARC 215 [10].
- **CSES Problem Set** [11] is a fixed collection, grouped by topic from introductory problems to advanced graph and string algorithms, which reached 400 problems in May 2025 [12]. It pairs

naturally with Laaksonen's guide [1], since both come from Antti Laaksonen, and it is an excellent way to make sure no standard technique is missing.

- **TLX** [13], the platform of the Indonesian Olympiad in Informatics team (TOKI), has free courses in Indonesian, from "Pemrograman Kompetitif Dasar" (basic competitive programming) upwards, together with the archive of Indonesian olympiad problems. For readers preparing for OSN, it is the first stop.

Table 14.3 Where to practice after SPOJ, and what each judge is best for.

Judge	Best for	Language of problems	Contests
SPOJ	classical problems, long archive	English	occasional
Codeforces	rated rounds, huge tagged archive	English, Russian	several per month
AtCoder	clean beginner contests (ABC)	Japanese, English	weekly
CSES	a complete syllabus of standard techniques	English and others	none (fixed set)
TLX (TOKI)	Indonesian courses, OSN preparation	Indonesian, English	national training

14.7.2 Contests in 2026

For secondary-school students, the ladder is the Olympiad. **OSN** (Olimpiade Sains Nasional) in informatics is organized by the national talent and achievement center, Puspresnas, and its national round in 2026 took place in Malang from 14 to 20 September [14]. The best OSN participants join TOKI's training camps, from which Indonesia's team for the **International Olympiad in Informatics** is chosen. IOI 2026, the 38th, was held in Tashkent, Uzbekistan, from 9 to 16 August 2026 [15].

For university students, the ladder is the **ICPC**. In Indonesia the season starts with the **Indonesia National Contest (INC)**, organized by BINUS University as the online preliminary for Indonesian teams: in 2026 it runs on Sunday, 25 October, from 10:00 to 15:00 WIB, with ten to thirteen problems in five hours, in C, C++, Java or Python [16]. The best INC teams advance to the onsite **ICPC Asia Jakarta Regional Contest**, which also welcomes teams from other countries [17], and regional winners across Asia continue to the Asia Pacific championship and the **ICPC World Finals**. The 50th World Finals will be held in Dubai, United Arab Emirates, from 15 to 20 November 2026 [18], [19].

The format of ICPC contests matches the workflow of this chapter closely: three students share one computer, so steps 1 to 4 happen on paper while a teammate types; a wrong submission costs twenty minutes of penalty time, so step 7 pays for itself; and the team that has practiced a routine together has a large advantage over three strong individuals who have not.

14.7.3 Books to keep on the desk

Three books extend this one in different directions. Laaksonen's *Guide to Competitive Programming* [1], now in its third edition (2024), is compact and covers the standard techniques with proofs, and it is the companion of the CSES Problem Set. Halim, Halim and Effendy's *Competitive Programming 4* [2] is encyclopedic, organized by problem type, and full of references to judge problems. Cormen et al.'s *Introduction to Algorithms* [4] is the reference for proofs and analysis whenever a technique needs a deeper understanding than a contest book gives. In Indonesian, TOKI's book and course *Pemrograman Kompetitif Dasar* [13] is the natural starting point.

14.7.4 A twelve-week plan

Skill in competitive programming grows with regular practice more than with long sessions. **Table 14.4** is a plan for the twelve weeks after this book, for about eight hours a week. Each week has a topic, a source of problems and one habit to practice deliberately.

Table 14.4 A twelve-week practice plan after this book, with one habit to practice deliberately in each block.

Weeks	Topic	Where	Habit to practice
1–2	Implementation, sorting, simulation	AtCoder ABC A–C, CSES introductory	Step 1: restate before coding
3–4	Binary search, two pointers, greedy	CSES sorting and searching, Codeforces Div. 3/4	Step 2: budget from constraints
5–6	Dynamic programming	CSES dynamic programming, AtCoder Educational DP	Step 3: write the recurrence first
7–8	Graphs	CSES graph algorithms, SPOJ problems of Chapter 12	Step 4: a brute force for every problem
9–10	Number theory, strings, geometry	CSES mathematics and string algorithms, SPOJ problems of Chapter 13	Step 7: stress test before submitting
11–12	Mixed contests under time pressure	Virtual contests on Codeforces and AtCoder, past INC problem sets	The whole workflow under the clock

After every contest, *upsolve*: solve the problems you could not solve during the contest, with the editorial if needed. Most of the learning happens there. Keep a notebook, paper or a text file, with one line per mistake you made and how you found it; after a few months it becomes your personal version of the “Common Mistakes” sections of this book.

14.8 Using It for Real

The workflow is not only for judges. Every time a program must be *right*, the same steps apply. A program that computes the laundry’s delivery schedule, the school’s timetable or an orchard’s fertilizer order can be tested exactly as PPATH was: write the obvious slow version for tiny cases, generate random cases, and compare. A module whose input space is small, like ONEZERO’s 20,000 values of n, can be checked exhaustively before it is trusted. And when a real system misbehaves only on some inputs, the shrinking technique of Section 14.4.3 turns a long log into a two-line reproduction that anyone can debug.

For a team preparing for INC or the Jakarta regional, the most useful thing to build from this book is a **team notebook**: a few pages with your own tested versions of the templates you need most, such as the union-find and Dijkstra of Chapter 12, modular exponentiation, Miller–Rabin, the cross product and the prefix function of Chapter 13, and a stress-test script like the one in the `tests` folders of this book’s code. Check the contest rules for what printed material is allowed, type every template yourself, and stress-test it once more before it goes into the notebook.

14.9 Common Mistakes, and How to Avoid Them

Coding before understanding. Skipping step 1 is the most expensive shortcut. If you cannot restate the task in two sentences, you are not ready to type.

Trusting the samples. Samples are chosen to explain the format, not to test your program; ABCPATH’s single-grid sample hides the reset bug completely. Write your own tests, and generate random ones.

A brute force that shares code with the fast solution. If both use the same helper, a bug in the helper makes them agree on the wrong answer. Write the brute force in the most different, most obvious way you can.

Stress-testing only one test case per file. State that leaks from one test case to the next is invisible when each file has one test. Generate several per file.

Ignoring the size of the answer. ONEZERO's input is small and its answer is huge. Estimate the output size in step 2 as carefully as the input size.

Changing several things after a failed verdict. Change one thing, test it, and only then submit again; otherwise you learn nothing from the next verdict.

Not upsolving. A contest in which you solved two problems and then read the editorials of the rest teaches more than three contests in which you only solved two.

14.10 Chapter Summary

- A routine of eight steps, **Figure 14.1**, turns the techniques of this book into Accepted verdicts: read, budget, idea, brute force, fast solution, own tests, stress test, submit.
- **PPATH** is BFS on an implicit graph of four-digit primes with a sieve for primality; the brute force is an independent Bellman–Ford relaxation.
- **ABCPATH** is memoized search, a DP on the letters; a planted bug (a memo reset only once) passes the samples, and a stress test with several grids per file catches it and shrinks it to a two-line input.
- **ONEZERO** is BFS over remainders, Equation (14.3); the tempting BFS over the numbers themselves overflows, and an exhaustive checker over all 20,000 values of n proves the correct program right and the wrong one wrong.
- Every verdict other than Accepted points to a small set of causes, **Table 14.2**.
- Next steps: Codeforces, AtCoder, CSES and TLX for practice; OSN and IOI for school students; INC, the ICPC Asia Jakarta regional and the ICPC World Finals (Dubai, November 2026) for university students; Laaksonen, Halim and Cormen for reading; and a twelve-week plan, **Table 14.4**, with deliberate habits.

14.11 Exercises

1. For PPATH, explain why the digit loop for the thousands place must start at 1, and give an input on which a program that starts it at 0 prints a wrong answer.
2. Using the neighbors of 1009 listed in Section 14.3.3, explain why the answer for $1009 \to 1019$ is 1 and for $1009 \to 1033$ is 2.
3. Write the brute force for ONEZERO described in Section 14.5.4 in five lines of Python, and use it to find the answers for $n = 3$ and $n = 6$.
4. For ONEZERO with $n = 3$, draw the BFS over remainders as in **Figure 14.2** and read off the answer.
5. Prove that for every n the answer to ONEZERO has at most n digits.
6. In ABCPATH, what goes wrong if the recursion forgets the bounds check for neighbors outside the grid? Why might the program still pass many tests?
7. Give a grid of size 1×3 on which a program that searches only from the *first* `A` it finds, instead of from every `A`, prints a wrong answer.

8. A friend's solution to a problem gets Time Limit Exceeded; locally it runs in 0.3 seconds on the sample. List three things to check, in order.
9. Plan your first two weeks from **Table 14.4**: pick three concrete problems from AtCoder or CSES for each week, and for each one write which step of the workflow you expect to be hardest.
10. Write a checker (not a brute force) for PPATH that verifies a claimed answer k for a pair of primes, given also the path. What must it check?

14.12 Solutions to the Exercises

1. The statement allows only four-digit primes, and a number whose first digit is 0 is not a four-digit number. A program whose loop starts at 0 for the thousands place also accepts 0013 = 13, 0017 = 17 and every other prime below 1,000 as vertices, because the sieve says they are prime. Then shortcuts appear. For the pair 1013 → 2017 the correct answer is 4, for example 1013 → 9013 → 9011 → 2011 → 2017, but the faulty program finds 1013 → 0013 → 0017 → 2017 and prints 3. The stress test with random pairs finds such a pair quickly, because the brute force of Section 14.3.4 only relaxes four-digit primes.

2. 1019 appears in the list of prime neighbors of 1009, so it is reached in one change. 1033 differs from 1009 in two digits (tens and units), so it needs at least two changes; the path 1009 → 1039 → 1033 changes the tens digit and then the units digit, and both 1039 and 1033 are prime, so the answer is 2.

3.

```
def brute(n):
    k = 1
    while int(bin(k)[2:]) % n:
        k += 1
    return int(bin(k)[2:])
```

$k = 1,2,3,\ldots$ gives 1,10,11,100,101,110,111, ... For $n = 3$ the first multiple is 111; for $n = 6$ it is 1110.

4. Start at remainder 1 mod 3 = 1. Append 0: 10 mod 3 = 1, already seen. Append 1: 11 mod 3 = 2, new. From 2: append 0 gives 20 mod 3 = 2, seen; append 1 gives 21 mod 3 = 0: done. The path is start 1, append 1, append 1: the answer is 111.

5. Consider the numbers $R_j = 11\ldots1$ (j ones) for $j = 1, \ldots, n+1$. There are $n+1$ of them and only n possible remainders modulo n, so two, say R_i and R_j with $i < j$, have the same remainder. Then $R_j - R_i = 11\ldots100\ldots0$ ($j - i$ ones followed by i zeros) is divisible by n and is written with 0s and 1s. It has $j \le n+1$ digits. Sharpening the count: if some R_j with $j \le n$ is itself divisible by n we are done with at most n digits; otherwise the n numbers $R_1, \ldots, R_n$ have their remainders among the $n-1$ non-zero values, so two coincide with $j \le n$, and the difference has $j \le n$ digits. The smallest answer is no longer than this one.

6. Without the bounds check, the program reads `grid[-1][...]` or `grid[r][w]`, memory outside the rows it filled. With the global array of size 52×52 the out-of-range cells often contain zero bytes or letters left from a previous, larger grid, so the program may print correct answers most of the time, which is exactly why the bug is dangerous. AddressSanitizer reports a read before the first row, which lies outside the array, but not a read of a cell beyond the current grid's width or height that is still inside the 52×52 array; only a careful bounds check, or a stress test with grids of different sizes one after another, reveals that.

7. Use `AAB`: the first `A` at column 0 is not adjacent to the `B` at column 2, so a search from the first `A` only gives 1, while the `A` at column 1 gives 2. The correct answer is 2.

8. First, recompute the complexity budget with the largest values the statement allows, including the number of test cases, and time the program locally on a generated maximum input; a sample is not a maximum. Second, check input and output speed (Chapter 4): unsynchronized `cin`, `endl` in a loop, or Python `input()` for a million lines. Third, look for hidden quadratic work: `erase` from the front of a vector, string concatenation in a loop, a linear search inside a loop, or a data structure reset with a full `memset` for each of many small test cases.

9. A sample plan: week 1, AtCoder ABC problems A, B and C from one recent contest, and CSES "Weird Algorithm", "Missing Number" and "Repetitions"; the hardest step is usually step 1 for problem C, where the statement is longest. Week 2, CSES "Increasing Array", "Permutations" and "Number Spiral", and one Codeforces Div. 4 round; for "Number Spiral" the hardest step is step 2, because the coordinates go up to 10^9 and a simulation is impossible. The point of the exercise is the second column: naming in advance where you expect difficulty makes you slow down at the right place.

10. Given two primes a, b and a claimed path $a = x_0, x_1, \ldots, x_k = b$, the checker must verify that every x_i is a four-digit number with a non-zero first digit and is prime, that consecutive x_i differ in exactly one digit, and that x_0 and x_k are the given numbers. That proves a path of length k exists. To prove that no shorter path exists, the checker still needs a search (or a trusted distance table); this is the difference between checking feasibility, which is easy, and checking optimality, which is usually as hard as solving the problem. ONEZERO's checker could prove optimality cheaply only because the set of reachable remainders per number of digits is small.

References

[1] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

[2] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*. Singapore: Lulu, 2020.

[3] SPOJ, "PPATH — Prime Path," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/PPATH/

[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] SPOJ, "ABCPATH — ABC Path," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/ABCPATH/

[6] SPOJ, "ONEZERO — Ones and zeros," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/ONEZERO/

[7] Codeforces, accessed Sep. 2026. [Online]. Available: https://codeforces.com/

[8] AtCoder, "Contests," accessed Sep. 2026. [Online]. Available: https://atcoder.jp/contests/

[9] AtCoder, "AtCoder Beginner Contest 473," Aug. 29, 2026. [Online]. Available: https://atcoder.jp/contests/abc473

[10] AtCoder, "Changes to ARC in 2026," 2026. [Online]. Available: https://atcoder.jp/posts/2026ARC_en

[11] CSES, "CSES Problem Set," accessed Sep. 2026. [Online]. Available: https://cses.fi/problemset/

[12] CSES, "CSES Problem Set May 2025 Update," May 2025. [Online]. Available: https://cses.fi/blog/text/3613

[13] Tim Olimpiade Komputer Indonesia, "TLX — Courses: Pemrograman Kompetitif Dasar," accessed Sep. 2026. [Online]. Available: https://tlx.toki.id/courses/competitive-1

[14] Tim Olimpiade Komputer Indonesia, "OSN 2026 Bidang Informatika," Sep. 2026. [Online]. Available: https://peringkat.toki.id/osn-2026/

[15] IOI 2026 Organizing Committee, "International Olympiad in Informatics 2026, Tashkent," accessed Sep. 2026. [Online]. Available: https://ioi2026.uz/

[16] BINUS University, "ICPC 2026: Indonesia National Contest (INC)," accessed Sep. 2026. [Online]. Available: https://competition.binus.ac.id/icpc2026/index/aboutinc

[17] ICPC Foundation, "The 2026 ICPC Asia Jakarta Onsite Regional Contest," accessed Sep. 2026. [Online]. Available: https://icpc.global/regionals/finder/Jakarta

[18] ICPC Foundation, "The 2026 ICPC World Finals Dubai hosted by Dubai Chamber of Digital Economy," accessed Sep. 2026. [Online]. Available: https://worldfinals.icpc.global/2026/about/

[19] Codeforces, "The 2026 ICPC World Finals: Dubai, U.A.E. November 15–20, 2026," accessed Sep. 2026. [Online]. Available: https://codeforces.com/blog/entry/149884