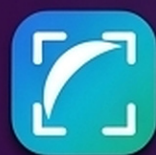


Informatics Olympiad (OSN)

Road to National Champion — 15 SPOJ Case Studies



Profitina

Irfan Subakti

profitina.com

Informatics Olympiad (OSN): Road to National Champion

15 SPOJ Case Studies

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	xvi
About this Public Edition	xvi
Who this book is for	xvii
One language, tested programs	xvii
What is in it	xvii
How to read it	xviii
About the author	xviii
Acknowledgments	xix
 CHAPTER 1 The Champion’s Playbook: Contest Strategy and Careful Reading —	
HOTLINE	1
1.1 A Letter to a Future Finalist	1
1.2 OSN Informatics in 2026	2
1.2.1 The stages	2
1.2.2 How the national round is scored	2
1.2.3 After the national round: TOKI and the IOI	3
1.2.4 The contest environment	3
1.3 Your Workbench in Ten Minutes	3
1.4 Reading Bounds Like a Problem Setter	4
1.5 The Contest Day, Minute by Minute	4
1.5.1 Minutes 0 to 10: map everything	4
1.5.2 Points per minute	5
1.5.3 The shape of a good day	5
1.5.4 The 45-minute rule	5
1.5.5 Before the day	5
1.6 Subtasks: The Mathematics of Medals	6
1.7 Warm-Up: Cutting a Line into Words	6
1.8 The Debug Protocol and the Stress Test	7
1.9 HARD Case Study: HOTLINE	8
1.9.1 The problem in our own words	8
1.9.2 A sample dialogue	9
1.9.3 Modeling: facts as keys	9
1.9.4 The full solution	10
1.9.5 Line by line	13
1.9.6 Traps	14
1.9.7 Stress test against a brute force	14
1.10 Contest-Day Tactics	16
1.11 Summary	16
1.12 Exercises	16
1.13 Solutions to the Exercises	17
References	17
 CHAPTER 2 Segment Trees and Mergeable Answers — GSS1	19
2.1 A Cooperative’s Ledger	19
2.2 The Idea in Plain Words	19
2.3 Warm-Up Problems	20
2.3.1 Range sums without updates	20
2.3.2 The best block of the whole array	21
2.4 The Segment Tree	21
2.4.1 Shape and numbering	21

2.4.2 Build in linear time.....	22
2.4.3 Query in logarithmic time	22
2.5 Designing the Node.....	23
2.5.1 The closure recipe	23
2.5.2 Four fields for the best block.....	23
2.5.3 Why the formulas are right	23
2.5.4 Working the example by hand	24
2.5.5 The same recipe, other questions.....	24
2.6 HARD Case Study: GSS1	25
2.6.1 The problem in our own words.....	25
2.6.2 Sample input and output.....	25
2.6.3 Design	25
2.6.4 The full solution.....	26
2.6.5 Line by line	27
2.6.6 Traps.....	27
2.6.7 Stress test.....	28
2.7 When Another Tool Is Better	29
2.8 Contest-Day Tactics	29
2.9 Summary	29
2.10 Exercises.....	29
2.11 Solutions to the Exercises	30
References.....	30
CHAPTER 3 Persistent Segment Trees and Order Statistics — MKTHNUM.....	31
3.1 The Market Inspector’s Median.....	31
3.2 The Idea in Plain Words	31
3.3 Warm-Up Problems.....	32
3.3.1 Coordinate compression	32
3.3.2 Subtracting two prefix counts	32
3.4 A Segment Tree over Values	33
3.5 Persistence by Path Copying	34
3.5.1 Copy the path, share the rest.....	34
3.5.2 How much memory	34
3.5.3 Why sharing is safe	35
3.6 Subtracting Two Versions.....	35
3.7 HARD Case Study: MKTHNUM.....	36
3.7.1 The problem in our own words.....	36
3.7.2 Sample input and output.....	36
3.7.3 Design	36
3.7.4 The full solution.....	36
3.7.5 Line by line	38
3.7.6 Traps.....	38
3.7.7 Stress test.....	38
3.7.8 The cost in numbers	39
3.8 Relatives and Alternatives.....	40
3.9 Contest-Day Tactics	40
3.10 Summary	40
3.11 Exercises.....	41
3.12 Solutions to the Exercises	41
References.....	41
CHAPTER 4 Monotonic Stacks and Linear Array Techniques — HISTOGRAM.....	43
4.1 The Banner on Main Street.....	43
4.2 The Idea in Plain Words	43
4.2.1 Two slower roads to the same answer	44

4.3 Warm-Up Problems.....	44
4.3.1 Nearest smaller elements	44
4.3.2 Stock span	45
4.4 The Monotonic Stack.....	46
4.4.1 The invariant	46
4.4.2 Why a loop inside a loop is linear.....	46
4.4.3 Four variants, one pattern.....	47
4.5 One Pass: Answers at Pop Time.....	47
4.5.1 Why the single pass finds the maximum	48
4.6 HARD Case Study: HISTOGRAM.....	49
4.6.1 The problem in our own words.....	49
4.6.2 Sample input and output.....	49
4.6.3 Design	49
4.6.4 The full solution.....	49
4.6.5 Line by line	50
4.6.6 Traps.....	50
4.6.7 Stress test.....	51
4.7 From Histograms to Matrices.....	52
4.8 Contest-Day Tactics	52
4.9 Summary	52
4.10 Exercises.....	53
4.11 Solutions to the Exercises	53
References.....	53
CHAPTER 5 Living Data Structures: Point Updates in Segment Trees — GSS3	55
5.1 Corrections in the Ledger	55
5.2 The Idea in Plain Words	55
5.2.1 How little changes	56
5.3 Warm-Up: Range Sums with Updates.....	56
5.4 The Point Update.....	58
5.4.1 The algorithm	58
5.4.2 Why it is correct	58
5.4.3 Cost.....	58
5.4.4 A bottom-up alternative	59
5.5 A Practice Cycle That Imitates the Contest.....	59
5.5.1 What to record while practising	59
5.6 HARD Case Study: GSS3.....	60
5.6.1 The problem in our own words.....	60
5.6.2 Sample input and output.....	60
5.6.3 Design	60
5.6.4 The full solution.....	60
5.6.5 Line by line	62
5.6.6 Traps.....	62
5.6.7 Stress test	62
5.6.8 A bug caught in the act	63
5.7 Beyond Point Updates.....	64
5.8 Contest-Day Tactics	64
5.9 Summary	65
5.10 Exercises.....	65
5.11 Solutions to the Exercises	65
References.....	66
CHAPTER 6 State-Space Graphs and Dial's Algorithm — CHPSLL	67
6.1 A Hero, a Map and a Stopwatch.....	67
6.2 Seeing the Hidden Graph.....	67

6.2.1 States, actions and costs	67
6.2.2 The weapon ladder	68
6.2.3 Why Dijkstra works, in one paragraph	68
6.2.4 Dial's algorithm: a ring of buckets.....	69
6.2.5 A state space without a map	70
6.3 Warm-Up Problems.....	71
6.3.1 A knight on a board with forbidden squares.....	71
6.3.2 Breaking the fewest walls: 0-1 BFS	72
6.4 Case Study: CHIPSLL.....	73
6.4.1 The problem in our own words.....	73
6.4.2 A sample worked by hand	73
6.4.3 Designing the solution.....	73
6.4.4 The complete solution	74
6.4.5 Line by line	76
6.4.6 Traps.....	77
6.5 Stress Testing Against a Brute Force	77
6.6 Contest-Day Tactics.....	79
6.7 Chapter Summary	79
6.8 Exercises.....	80
6.9 Solutions to the Exercises	80
References.....	80
CHAPTER 7 Disjoint Sets, Kruskal's Algorithm and Sweeps — VILUNIT	82
7.1 Villages on a Map	82
7.2 The Tools from Zero.....	83
7.2.1 Disjoint-set union in five lines	83
7.2.2 Kruskal's algorithm and the cut property.....	83
7.2.3 Sweeping intervals.....	84
7.2.4 Why the DSU is fast enough	85
7.3 Warm-Up: Kruskal on a Small Graph	85
7.4 Case Study: VILUNIT	86
7.4.1 The problem in our own words.....	86
7.4.2 Sample input and output.....	87
7.4.3 Designing the solution.....	87
7.4.4 Tracing the second sample by hand	88
7.4.5 The complete solution	88
7.4.6 Line by line	90
7.4.7 Complexity, honestly	91
7.4.8 Traps.....	91
7.5 Stress Testing Against a Brute Force	92
7.6 Using the Idea Beyond VILUNIT.....	93
7.7 Contest-Day Tactics	93
7.8 Chapter Summary	94
7.9 Exercises.....	94
7.10 Solutions to the Exercises	94
References.....	95
CHAPTER 8 Euler Tours, Fenwick Trees and Binary Lifting — HBD.....	96
8.1 Collecting Donations Up the Family Tree.....	96
8.2 The Three Gears from Zero	96
8.2.1 Gear 1: the Euler tour turns subtrees into intervals	97
8.2.2 Gear 2: a Fenwick tree for prefix sums with updates	98
8.2.3 Path sums as differences	98
8.2.4 A Fenwick tree by hand.....	98
8.2.5 Gear 3: binary lifting.....	99

8.3 Warm-Up: The Three Gears on a Small Tree	99
8.4 Case Study: HBD	101
8.4.1 The problem in our own words	101
8.4.2 Sample input and output	101
8.4.3 Designing the solution	102
8.4.4 Tracing one question	102
8.4.5 The complete solution	103
8.4.6 Line by line	104
8.4.7 Complexity and traps	105
8.5 Stress Testing Against a Brute Force	105
8.6 The Same Gears, Other Problems	107
8.7 Contest-Day Tactics	108
8.8 Chapter Summary	108
8.9 Exercises	108
8.10 Solutions to the Exercises	108
References	109
CHAPTER 9 Counting Distinct Objects with Dynamic Programming — DSUBSEQ	110
9.1 Distinct Words from One Word	110
9.2 Counting Without Double Counting	111
9.2.1 Subsequences	111
9.2.2 The state, as a sentence	111
9.2.3 Deriving the recurrence	111
9.2.4 Two tempting mistakes	112
9.2.5 A second derivation: counting by the last letter	112
9.2.6 Arithmetic modulo a prime	113
9.3 Warm-Up Problems	113
9.3.1 Counting ways: occurrences of ABC	113
9.3.2 Watching the recurrence work	114
9.3.3 BANANA, the classroom word	114
9.3.4 From ways to strings	114
9.4 Case Study: DSUBSEQ	115
9.4.1 The problem in our own words	115
9.4.2 Sample input and output	115
9.4.3 Designing the solution	115
9.4.4 The complete solution	116
9.4.5 Line by line	116
9.4.6 Why the modulus is harmless	117
9.4.7 Traps	117
9.5 Stress Testing Against a Brute Force	117
9.6 The Pattern in Other Problems	118
9.7 Contest-Day Tactics	119
9.8 Chapter Summary	119
9.9 Exercises	120
9.10 Solutions to the Exercises	120
References	121
CHAPTER 10 Bitmask and Profile Dynamic Programming — CFONISMG	122
10.1 Seating a Classroom	122
10.2 Sets as Numbers	123
10.2.1 Bit operations	123
10.2.2 The model: the travelling salesman	123
10.3 The Profile: a Grid's Cross-Section as State	125
10.3.1 Filling cell by cell	125
10.3.2 Warm-up: domino tilings	125

10.3.3 Encoding a profile as one number	126
10.3.4 Estimate before you code	126
10.3.5 Arrays or hash maps?	127
10.4 Case Study: CFONISMG.....	127
10.4.1 The problem in our own words	127
10.4.2 Sample input and output	128
10.4.3 Designing the solution	128
10.4.4 A tiny run by hand	128
10.4.5 The complete solution	129
10.4.6 Line by line.....	131
10.4.7 Traps.....	132
10.5 Stress Testing Against a Brute Force.....	132
10.6 Contest-Day Tactics	134
10.7 Chapter Summary.....	134
10.8 Exercises.....	135
10.9 Solutions to the Exercises	135
References.....	135
CHAPTER 11 CDQ Divide and Conquer and the Two-Dimensional LIS — LIS2	137
11.1 Where This Helps: Dewi's Training Log.....	137
11.2 The Technique from Zero.....	138
11.2.1 From one dimension to two	138
11.2.2 The idea: split by position	138
11.2.3 Why the order left, cross, right is compulsory	139
11.2.4 The cost	140
11.3 Warm-Up Problems	140
11.3.1 Inversions: the left half influences the right half	140
11.3.2 The one-dimensional LIS with a Fenwick tree of maxima.....	141
11.4 Case Study: LIS2	142
11.4.1 The problem in our own words	142
11.4.2 A sample by hand.....	143
11.4.3 Design.....	143
11.4.4 The complete solution	143
11.4.5 Line by line.....	145
11.4.6 Traps.....	145
11.5 Testing It: CDQ Against the Quadratic DP	146
11.6 Contest-Day Tactics	147
11.6.1 CDQ over time: updates and queries.....	148
11.7 Chapter Summary.....	148
11.8 Exercises.....	148
11.9 Solutions to the Exercises	149
References.....	149
CHAPTER 12 Meet in the Middle: Halving the Exponential — ABCDEF	150
12.1 Where This Helps: Bu Sari's Parcels.....	150
12.2 The Technique from Zero.....	151
12.2.1 The shape to look for	151
12.2.2 The recipe.....	151
12.2.3 Time and memory, estimated before coding	152
12.2.4 A run by hand.....	152
12.3 Warm-Up Problems	153
12.3.1 Counting pairs with a given sum	153
12.3.2 Subset sums in a window, $n \leq 40$	153
12.4 Case Study: ABCDEF	154
12.4.1 The problem in our own words	154

12.4.2 From one equation to two sides.....	155
12.4.3 Where to enforce $d \neq 0$	155
12.4.4 The samples.....	155
12.4.5 The complete solution	155
12.4.6 Line by line.....	156
12.4.7 Numbers that save or bury you	157
12.5 Testing It: Meet in the Middle Against n^6	157
12.6 Contest-Day Tactics	158
12.6.1 Relatives of meet in the middle	158
12.6.2 Common mistakes	159
12.7 Chapter Summary.....	159
12.8 Exercises.....	159
12.9 Solutions to the Exercises	160
References.....	161
CHAPTER 13 Suffix Arrays and LCP: Counting Distinct Substrings — DISUBSTR.....	162
13.1 Where This Helps: Words in a Batik Pattern	162
13.2 The Technique from Zero.....	163
13.2.1 Suffix array and LCP array	163
13.2.2 Building the suffix array by prefix doubling	163
13.2.3 The LCP array in linear time: Kasai's algorithm.....	164
13.2.4 From the LCP array to the number of distinct substrings	165
13.2.5 Kasai by hand on ABABA.....	165
13.3 Warm-Up Problems	166
13.3.1 The slow suffix array, as a table	166
13.3.2 Pattern search with the suffix array	166
13.4 Case Study: DISUBSTR.....	167
13.4.1 The problem in our own words	167
13.4.2 Sample.....	167
13.4.3 Design.....	168
13.4.4 The complete solution	168
13.4.5 Line by line.....	170
13.5 Testing It: Suffix Array Against a Set of Strings	170
13.6 Contest-Day Tactics	171
13.6.1 Two strings at once	171
13.6.2 Choosing the right tool	172
13.6.3 Common mistakes	172
13.7 Chapter Summary.....	172
13.8 Exercises.....	172
13.9 Solutions to the Exercises	173
References.....	174
CHAPTER 14 Finite Differences, Interpolation and Modular Arithmetic — BATTLECRY.....	175
14.1 Where This Helps: Counting Without Listing.....	175
14.2 The Technique from Zero.....	175
14.2.1 Finite differences: the polynomial stethoscope.....	175
14.2.2 Lagrange interpolation: from points to a polynomial.....	176
14.2.3 Horner's rule and modular arithmetic	177
14.2.4 Why placements give polynomials.....	177
14.2.5 Fast exponentiation by hand.....	178
14.2.6 How many values do we need?.....	178
14.3 Warm-Up Problems	178
14.3.1 The difference table	178
14.3.2 Lagrange evaluation modulo a prime	179

14.3.3 The whole pipeline on a smaller problem	180
14.4 Case Study: BATTLECRY	181
14.4.1 The problem in our own words	181
14.4.2 Understanding checkmate precisely	181
14.4.3 The pipeline, step by step	181
14.4.4 Sample	182
14.4.5 The complete solution	182
14.4.6 Line by line	183
14.5 Testing It: the Formula Against the Chessboard	184
14.6 Contest-Day Tactics	186
14.6.1 When the answer is not a polynomial	187
14.6.2 Common mistakes	187
14.7 Chapter Summary	188
14.8 Exercises	188
14.9 Solutions to the Exercises	188
References	189
CHAPTER 15 Maximum Flow with Dinic's Algorithm — FASTFLOW	190
15.1 Where This Helps: Water for the Orchard	190
15.2 The Technique from Zero	190
15.2.1 Flows, capacities and cuts	190
15.2.2 Residual graphs and augmenting paths	191
15.2.3 Edmonds–Karp: shortest augmenting paths	191
15.2.4 Dinic: many paths per BFS	191
15.2.5 Why each phase makes the sink farther away	193
15.3 Warm-Up: Edmonds–Karp on a Matrix	193
15.4 Case Study: FASTFLOW	195
15.4.1 The problem in our own words	195
15.4.2 Sample	195
15.4.3 Design decisions	195
15.4.4 The complete solution	195
15.4.5 Line by line	197
15.5 Testing It: Dinic Against the Minimum Cut	198
15.6 Contest-Day Tactics	199
15.6.1 A modeling example, worked	199
15.7 Your Road to the National Podium	200
15.7.1 What the fifteen chapters gave you	200
15.7.2 A sixteen-week plan	200
15.7.3 Daily and weekly habits	200
15.7.4 On the day	201
15.7.5 A readiness checklist	201
15.8 Chapter Summary	201
15.9 Exercises	202
15.10 Solutions to the Exercises	202
References	203

List of Figures

Figure 1.1 A five-hour contest day: map, bank the easy subtasks, attack, finish the other problems, and freeze the last half hour; the dashed lines are check-points for the 45-minute rule.	5
Figure 2.1 The segment tree of 2, -5, 4, -1, 3, -6, 5, 1; the query [2,5] visits the red nodes and combines the three amber nodes [2], [3,4] and [5].	22
Figure 3.1 Path copying: inserting value 3 into version 1 creates only the three amber nodes on the path to leaf [3]; the dashed edges point to nodes shared with version 1.	34
Figure 4.1 The histogram 2, 1, 4, 5, 1, 3, 3 and its largest rectangle; the rectangle is found when bar 4 pops bar 2 from the stack.	48
Figure 5.1 The tree of 1, 2, -3, 4 after the update “0 3 -3”: the amber leaf is replaced, the two red nodes on its path are recombined, and the other nodes keep their values.	56
Figure 6.1 A 5×6 map with entry costs (large numbers) and the distances d computed by Dial’s algorithm; the shaded squares form a cheapest route of cost 17.	70
Figure 7.1 Five villages and their projections (colored bars on the axes); the link costs 0, 1 and 2 come from which projections overlap.	87
Figure 8.1 A tree with entry and exit times $[tin, tout]$; in entry order, the subtree of node 5 fills the single interval of positions 4 to 7.	97
Figure 9.1 From AB to ABA: appending A to each old subsequence creates one duplicate, the string A, which the previous A already produced.	112
Figure 10.1 Filling a 3×4 grid in reading order: while cell 6 is filled, only the last four cells (the profile) can still influence the future.	125
Figure 11.1 CDQ on eight points. (a) The range is cut by index; the left half is finished before it influences the right half. (b) For the right point $p = (7,6)$ every left point with a smaller x has been inserted into the Fenwick tree, which then answers “largest dp with $y < 6$ ”.	139
Figure 12.1 Two sorted lists and their handshakes: each line is one counted pair, and equal values form runs whose sizes multiply.	152
Figure 13.1 The sorted suffixes of ABABA. The first $lcp[k]$ characters of each suffix (grey) repeat its upper neighbor; only the remaining prefixes (teal) are new substrings, $15 - 6 = 9$ of them.	163
Figure 14.1 The finite-difference table of the BATTLECRY counts for $N = 3$ to 10. Each row holds the differences of the row above; row 5 is constant, so the counts follow a polynomial of degree 5.	176
Figure 15.1 The FASTFLOW sample as an undirected graph with capacities and BFS levels. The dashed self-loop is discarded; the edge 2-3 lies inside level 1 and is not used in the first phase.	192

List of Tables

Table 1.1 The road from a school nomination to the international team, as described by TOKI and the 2026 OSN guide.	2
Table 1.2 Reading the bounds: the largest input tells you which complexities survive a one-second limit.	4
Table 4.1 The four nearest-neighbour questions and the pop condition that answers each in one left-to-right pass.	47
Table 4.2 The single-pass algorithm on 2, 1, 4, 5, 1, 3, 3; every candidate area is computed at the moment its bar is popped.	48
Table 5.1 A two-hour practice routine with the rhythm of a contest day.	59
Table 6.1 The shortest-path ladder: the shape of the weights chooses the algorithm; D is the largest distance found.	68
Table 10.1 Set operations on bitmasks; the last loop visits every non-empty subset of m	123
Table 11.1 Running time of LIS2.cpp on the largest inputs.	147

Table 12.1 The ranges in ABCDEF: the values fit in 32 bits, the count does not	157
Table 12.2 Running time of ABCDEF .cpp at the maximum size.....	158
Table 13.1 Running time of DISUBSTR .cpp at the maximum size.	171
Table 15.1 Running time of FASTFLOW .cpp on inputs of the maximum size.....	199
Table 15.2 A sixteen-week training plan before a national round.	200
Table 15.3 A readiness checklist for the weeks before a national round.	201

List of Listings

Listing 1.1 tokenize .cpp: split each line into words and report the first end mark.....	6
Listing 1.2 stress .sh: compile both programs, then compare them on random inputs until they differ.....	8
Listing 1.3 HOTLINE .cpp: facts as keys, incremental contradiction checks, and one function per kind of question.....	10
Listing 1.4 gen .py: random HOTLINE dialogues from a four-person, three-verb vocabulary.	14
Listing 2.1 prefix_sum .cpp: static range sums in constant time per question.....	20
Listing 2.2 kadane .cpp: the best non-empty block of the whole array in one pass.....	21
Listing 2.3 GSS1 .cpp: the four-field segment tree; build in $O(N)$, each question in $O(\log N)$	26
Listing 2.4 brute .cpp: every block of every range, $O(N^2)$ per question.....	28
Listing 2.5 gen .py: random GSS1 inputs at three value scales.	28
Listing 3.1 compress .cpp: sort, remove duplicates, and replace every value by its rank.	32
Listing 3.2 count_prefix.cpp: counting values at most x in a range by subtracting two prefix count tables.....	32
Listing 3.3 MKTHNUM .cpp: persistent segment tree over compressed values; each question walks down two versions.	36
Listing 3.4 brute .cpp: sort a copy of each range, $O(n \log n)$ per question.	38
Listing 3.5 gen .py: random MKTHNUM inputs, with and without duplicates.....	39
Listing 4.1 nearest_smaller .cpp: nearest strictly smaller element on each side, two passes with a stack	44
Listing 4.2 stock_span .cpp: stock spans with a decreasing stack of (price, span) pairs.	45
Listing 4.3 HISTOGRAM .cpp: one pass with a monotonic stack and a height-0 sentinel.....	49
Listing 4.4 brute .cpp: every pair of first and last bar, with a running minimum height.....	51
Listing 4.5 gen .py: random HISTOGRAM inputs, several test lines ending with 0.	51
Listing 5.1 sum_update .cpp: a segment tree for range sums with point assignment.	56
Listing 5.2 GSS3 .cpp: the GSS1 tree with a point update; every operation costs $O(\log N)$	60
Listing 5.3 brute .cpp: updates on a plain array and every block of every range.	62
Listing 5.4 gen .py: random GSS3 inputs mixing updates and questions.	63
Listing 6.1 knight .cpp: breadth-first search over the 64 knight states, with forbidden squares.	71
Listing 6.2 zero_one_bfs .cpp: 0-1 BFS with a double-ended queue on a map of free squares and walls.	72
Listing 6.3 CHIPSLL .cpp: Dial's algorithm with 64 circular buckets on a flat grid.....	74
Listing 6.4 brute .cpp: the reference solution with a binary heap.	77
Listing 6.5 gen .py: random small inputs for CHIPSLL.....	78
Listing 6.6 stress .sh: compile, generate, compare, stop at the first mismatch.	79
Listing 7.1 interval_groups .cpp: grouping intervals by a sweep with the farthest right end.....	84
Listing 7.2 kruskal .cpp: Kruskal's algorithm with the five-line DSU.	85
Listing 7.3 VILUNIT .cpp: Kruskal by weight classes, with pass 0 and two sweeps.	88
Listing 7.4 brute .cpp: all pairs, explicit weights 0, 1, 2, and Kruskal.....	92

Listing 7.5 <code>gen.py</code> : random small villages for VILUNIT.	92
Listing 8.1 <code>tree_gears.cpp</code> : Euler tour, Fenwick path sums and binary lifting on nine nodes.	99
Listing 8.2 <code>HBD.cpp</code> : Euler tour, Fenwick tree and binary lifting composed.	103
Listing 8.3 <code>brute.cpp</code> : the walk-up reference for HBD.	105
Listing 8.4 <code>gen.py</code> : random small trees and operations for HBD.	106
Listing 9.1 <code>abc_count.cpp</code> : counting occurrences of a pattern as a subsequence.	113
Listing 9.2 <code>dsub_table.cpp</code> : the DSUBSEQ table for ABAB, step by step.	114
Listing 9.3 <code>DSUBSEQ.cpp</code> : distinct subsequences with the last-occurrence correction.	116
Listing 9.4 <code>brute.cpp</code> : listing every subsequence into a set.	117
Listing 9.5 <code>gen.py</code> : random short strings over small alphabets.	118
Listing 10.1 <code>tsp_bitmask.cpp</code> : the Held-Karp dynamic program for five cities.	124
Listing 10.2 <code>domino_profile.cpp</code> : counting domino tilings with a one-bit-per-column profile.	125
Listing 10.3 <code>CFONISMG.cpp</code> : bitmask of seated students plus a base-17 profile, stored in a hash map.	129
Listing 10.4 <code>brute.cpp</code> : backtracking over every seating plan.	132
Listing 10.5 <code>gen.py</code> : random small classrooms for CFONISMG.	134
Listing 11.1 <code>inversions.cpp</code> : counting inversions with merge sort, the ancestor of CDQ.	140
Listing 11.2 <code>lis_fenwick.cpp</code> : the classic LIS in $O(n \log n)$ with a Fenwick tree of prefix maxima.	141
Listing 11.3 <code>LIS2.cpp</code> : the two-dimensional LIS in $O(n \log^2 n)$ with CDQ divide and conquer.	143
Listing 11.4 <code>brute.cpp</code> : the $O(n^2)$ dynamic program used as the reference.	146
Listing 11.5 <code>gen.py</code> : random small inputs with many ties.	146
Listing 11.6 <code>stress.sh</code> : the stress-test driver.	146
Listing 12.1 <code>two_sum_count.cpp</code> : counting pairs with a given sum by sorting and two binary searches.	153
Listing 12.2 <code>subset_mitm.cpp</code> : counting subsets with a sum in $[A, B]$ by meet in the middle.	154
Listing 12.3 <code>ABCDEF.cpp</code> : counting the tuples of Equation (12.4) by meet in the middle.	155
Listing 12.4 <code>brute.cpp</code> : the n^6 reference.	157
Listing 12.5 <code>gen.py</code> : small random sets for ABCDEF.	157
Listing 13.1 <code>naive_sa.cpp</code> : the suffix array and LCP array computed the slow way, printed as a table.	166
Listing 13.2 <code>pattern_count.cpp</code> : counting occurrences of a pattern by binary search on the suffix array.	167
Listing 13.3 <code>DISUBSTR.cpp</code> : distinct substrings with a suffix array (prefix doubling) and Kasai's LCP.	168
Listing 13.4 <code>brute.cpp</code> : the set-of-substrings reference.	170
Listing 13.5 <code>gen.py</code> : random strings over a small and a mixed alphabet.	171
Listing 14.1 <code>diff_table.cpp</code> : the finite-difference table and the detected degree.	178
Listing 14.2 <code>lagrange.cpp</code> : evaluating the interpolating polynomial at a new point without computing coefficients.	179
Listing 14.3 <code>BATTLECRY.cpp</code> : the degree-5 formula evaluated with Horner's rule and a modular inverse.	182
Listing 14.4 <code>brute.cpp</code> : counting checkmates by trying every position.	184
Listing 14.5 <code>gen.py</code> : random small board sizes.	186
Listing 15.1 <code>edmonds_karp.cpp</code> : maximum flow by shortest augmenting paths on a capacity matrix.	193
Listing 15.2 <code>FASTFLOW.cpp</code> : Dinic's algorithm with level graphs and current arcs, $O(V^2E)$ in the worst case.	195
Listing 15.3 <code>brute.cpp</code> : the minimum cut by trying every set S	198
Listing 15.4 <code>gen.py</code> : small random undirected multigraphs.	198

List of Pseudocode

Pseudocode 6.1 Dial's algorithm with $C + 1$ circular buckets; line 6 skips copies left behind when a node's distance improved.....	69
Pseudocode 11.1 CDQ divide and conquer for the two-dimensional LIS.	139
Pseudocode 12.1 Counting matching pairs by meet in the middle.....	151
Pseudocode 13.1 Building the suffix array by prefix doubling.	164
Pseudocode 15.1 Dinic's algorithm with level graphs and current arcs.	192

List of Equations

(1.1)	4
(1.2)	5
(1.3)	9
(2.1)	20
(2.2)	21
(2.3)	22
(2.4)	22
(2.5)	23
(2.6)	23
(2.7)	23
(2.8)	23
(3.1)	32
(3.2)	33
(3.3)	34
(3.4)	35
(4.1)	43
(4.2)	47
(4.3)	47
(4.4)	52
(5.1)	58
(5.2)	58
(5.3)	60
(5.4)	64
(6.1)	68
(6.2)	69
(6.3)	69
(6.4)	73
(7.1)	83
(7.2)	83
(8.1)	97
(8.2)	98
(8.3)	98
(8.4)	98
(8.5)	99
(8.6)	102
(9.1)	111

(9.2)	111
(9.3)	112
(9.4)	113
(10.1).....	123
(10.2).....	124
(10.3).....	126
(10.4).....	128
(11.1).....	138
(11.2).....	138
(11.3).....	140
(12.1).....	151
(12.2).....	152
(12.3).....	154
(12.4).....	155
(13.1).....	163
(13.2).....	163
(13.3).....	163
(13.4).....	165
(14.1).....	176
(14.2).....	176
(14.3).....	177
(14.4).....	177
(14.5).....	177
(14.6).....	177
(15.1).....	190
(15.2).....	191
(15.3).....	192

Preface

It is twenty minutes past nine on the first day of the national round. Dimas has read all three problems. The first one he recognizes: an array, a stream of questions about ranges, a hundred thousand of each. He knows that a loop over every range will be too slow, and he has heard of a structure called a segment tree, but he has never written one without looking at a reference, and the reference is not allowed in the hall. He writes the loop, collects the points for the smallest subtask, and spends the rest of the morning on a tree that does not quite work. That evening, when the results appear, he learns that a bronze medal needed thirty more points — exactly the points of the subtask he could not finish.

This book is written so that the morning goes differently. The Olimpiade Sains Nasional in informatics, the OSN, asks students to solve algorithmic problems in C++ in five hours, with partial credit for every subtask they manage to solve. What separates the contestants who come home with a medal from those who do not is rarely talent alone. It is a small number of techniques known so well that they can be written from memory, the habit of reading bounds before choosing an algorithm, a routine that tests a program before the judge does, and a plan for the day that banks easy points before chasing hard ones.

The title promises a road to *national champion*, and it should be read the way Indonesian schools use the word *juara*: a student who wins a gold, silver or bronze medal in the national round is a national champion, and so, in a real sense, is one who earns an honourable mention or reaches the national final after beating thousands of others in the city and province rounds. No book can promise a single top spot, which depends on the day and on the other finalists as much as on you. This one aims to make you a finalist who scores well, and who knows why each point was earned.

The exercises come from SPOJ, the Sphere Online Judge, one of the oldest and broadest archives of contest problems. Fifteen of its classic hard problems — HOTLINE, GSS1, MKTHNUM, HISTOGRAM, GSS3, CHIPSELL, VILUNIT, HBD, DSUBSEQ, CFONISMG, LIS2, ABCDEF, DISUBSTR, BATTLECRY and FASTFLOW — are the case studies through which fifteen olympiad techniques are taught, one per chapter.

About this Public Edition

This is the Public Edition of the book, *15 SPOJ Case Studies*. Its text is the text of the full edition: every idea, every figure, every proof sketch, every warm-up program and every stress test is here, complete. The difference is in the fifteen case-study programs. Each of them shows about half of its code — the reading of the input, the data structures, the first half of the core algorithm and the output — and the other half, the part that turns the idea into an accepted program, is left for you. In the listing you will find it as a block that opens with the line `// <P50>` and closes with `// </P50>`. Right after the opening line, one to three comment lines say precisely what the missing part must do; then comes the line `// >>> YOUR TURN (Public Edition): write this part yourself, following the hint above. <<<`.

This is not a gap in the book; it is the most valuable exercise in it. Every missing part is explained in full in the chapter's prose, usually with a worked example and an equation, and the chapter's brute force, generator and stress script are complete, so you can check what you write the same way the programs of the book were checked. Write the part, run the stress test until it stays silent,

and then submit to SPOJ. When the judge answers *Accepted*, the program is yours in a way that a copied program never is — and in the contest hall, where no book is allowed, that is the only kind that counts. The full edition, *Conquering SPOJ*, prints all fifteen solutions complete.

Who this book is for

The book is written for secondary-school students preparing for the OSN in informatics, for their teachers and coaches, and for university students who want the same techniques for the ICPC or for technical interviews. It assumes that you can already write a working C++ program with loops, arrays, functions and the standard containers, and that you have met recursion and the idea of counting steps with the O notation. If that is not yet true, the companion book *Competitive Programming with SPOJ* starts from zero: it installs the workbench, teaches the reading of statements, the time and memory budgets and the stress test, and builds the basic toolkit of dynamic programming, greedy algorithms, binary search and graphs. This book begins where that one ends.

Each technique is still built from zero. Every chapter explains the idea in plain words, works it by hand on a small example with a drawing, argues why it is correct, counts its cost, and only then writes code. Warm-up problems with short programs come before the hard case study, so that the case study is the last step of a staircase and not a wall.

One language, tested programs

The national round is a C++ contest, and so is this book: every program is written in **C++17** and compiled with `g++ -std=c++17 -O2 -Wall -Wextra`, without a single warning, and also run under `-fsanitize=address,undefined`, which stops a program the moment it reads memory it does not own. The small input generators are written in Python 3, because they are shorter that way.

No program in this book is trusted because it passes the samples. Every case-study solution 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, and the test stops at the first disagreement. Every problem statement is retold in our own words, with its SPOJ code, so that you can read the original and submit your own solution.

What is in it

Chapter 1 is the champion’s playbook. It describes the OSN as it runs in 2026 — the city, province and national rounds, subtask scoring, the contest environment and the road through the TOKI training camp to the IOI — sets up the workbench in ten minutes, turns bounds into a budget of steps, plans the five hours of a contest day, and introduces the debug protocol and the stress test on HOTLINE, a problem that is mostly careful reading.

Chapters 2 to 5 are about arrays and ranges. Chapter 2 builds the segment tree and the idea of a mergeable answer, a node with four fields, on GSS1. Chapter 3 makes the tree persistent by path copying and answers “the k -th smallest in a range” by subtracting two versions, on MKTHNUM. Chapter 4 teaches the monotonic stack and the largest rectangle in a histogram, on HISTOGRAM. Chapter 5 brings the segment tree to life with point updates, on GSS3.

Chapters 6 to 8 are about graphs and trees. Chapter 6 finds the hidden graph in a state space and runs Dial's algorithm with circular buckets, on CHIPSL. Chapter 7 combines disjoint sets, Kruskal's algorithm and sweeps into a minimum spanning tree over weight classes, on VILUNIT. Chapter 8 composes an Euler tour, a Fenwick tree and binary lifting to answer questions about paths in a tree, on HBD.

Chapters 9 and 10 are about dynamic programming. Chapter 9 counts distinct subsequences without double counting, on DSUBSEQ. Chapter 10 turns sets into numbers and a grid's cross-section into a state, with bitmask and profile dynamic programming, on CFONISMG.

Chapters 11 to 15 are the advanced toolkit. Chapter 11 solves the two-dimensional longest increasing subsequence with CDQ divide and conquer, on LIS2. Chapter 12 halves an exponential search by meeting in the middle, on ABCDEF. Chapter 13 builds the suffix array and the LCP array and counts distinct substrings, on DISUBSTR. Chapter 14 finds a formula from a table of small answers with finite differences, interpolation and modular arithmetic, on BATTLECRY. Chapter 15 computes a maximum flow with Dinic's algorithm, on FASTFLOW, and ends with a road map from here to the national podium.

Every chapter follows the same path: a story and the problem it poses, the technique from zero with a drawing and a small example worked by hand, warm-up problems with short programs, the hard case study — the problem retold, its design, the complete program and its explanation line by line, the traps, and the stress test against a brute force — then contest-day tactics, a summary, exercises with complete solutions, and references.

The programs are collected, chapter by chapter, in the accompanying code folder, from [Chapter01](#) to [Chapter15](#). Each folder holds the case-study solution, the warm-up programs, the brute force `brute.cpp`, the generator `gen.py`, the stress script `stress.sh`, sample inputs, 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 a compiler open and a SPOJ account ready. Read Chapter 1 first, because it sets up the working habits that every later chapter relies on; after that, Chapters 2, 3 and 5 form one line about segment trees and are best read in order, while the other chapters can be read in any order, and each one names the earlier chapters it builds on. Before you read a case-study solution, read the retold problem and try it yourself for at least half an hour: 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, forget a modulo, reverse a comparison — and watch the stress test catch it. Then close the book and write the technique again from memory; in the contest hall, memory is all you will have.

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, where design and analysis of algorithms is one of his subjects, and

he has trained students for programming contests for many years. 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 trained with 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 Tim Olimpiade Komputer Indonesia and to the teachers and coaches across the country who prepare students for the olympiad every year. 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++ and GCC, 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.com

Chapter 1

The Champion's Playbook: Contest Strategy and Careful Reading — HOTLINE

A national olympiad in informatics is won long before the contest hall opens, and it is lost, surprisingly often, in the first ten minutes of the contest day. This chapter is about both halves. You will learn what the Olimpiade Sains Nasional (OSN) in informatics looks like in 2026, how its stages lead to the national training camp and to the International Olympiad in Informatics, and how its scoring rewards careful, partial progress. You will set up a small, reliable workbench, learn to read the bounds of a problem the way its setter wrote them, plan a five-hour day minute by minute, and build the one habit that separates calm contestants from panicking ones: the stress test, a slow but obviously correct program checked against your fast one on hundreds of random inputs. The chapter ends with a hard SPOJ problem, *HOTLINE*, whose difficulty is not an algorithm at all but reading: every word of the statement becomes a rule in the program, and one stray character is a wrong answer.

1.1 A Letter to a Future Finalist

Imagine Dimas, a second-year student at a public high school in a small town in East Java. He likes puzzles, he has written a few programs in C++, and last month his teacher told him that the school would send him to the first stage of the informatics olympiad. He is excited and a little afraid. He has read about students who “won OSN”, he has seen their photographs with medals, and he wonders whether someone from his school, without a private coach, can ever stand where they stood.

This book is written for Dimas, and for you. Its title promises a road to *national champion*, and it is worth saying clearly, and kindly, what that means. In Indonesian school culture the word *juara* is used generously: a student who brings home a gold, silver or bronze medal from the national round is a *juara nasional*, and so, in a real sense, is a student who earns an honourable mention or simply reaches the national final after beating thousands of others in the city and province rounds. The goal of this book is not a single first place, which depends on the day, on luck, and on the other finalists as much as on you. The goal is to make you a contestant who reaches the national round and scores well there, and who knows exactly why each point was earned.

That goal is realistic. The problems of the national round are hard, but they are made from a finite set of ideas: data structures that answer questions about ranges, dynamic programming, graph algorithms, careful counting, and the discipline to read a statement precisely. Each chapter of this book takes one of those ideas, builds it from nothing, and then uses it on a hard problem from the Sphere Online Judge (SPOJ), a free online judge with thousands of problems that has trained competitive programmers for two decades. The companion book *Competitive Programming with SPOJ* [1] covers the basics (the judge, input and output, complexity, sorting, the first data structures); this book assumes you have met those ideas and climbs from there.

1.2 OSN Informatics in 2026

1.2.1 The stages

The Olimpiade Sains Nasional is organized by the Pusat Prestasi Nasional (Puspresnas) of the Ministry of Primary and Secondary Education, and in informatics the problems, the syllabus and the contest system are prepared by the Tim Olimpiade Komputer Indonesia (TOKI). The official OSN informatics site describes three levels [2]: the city or regency round (OSN-K), the provincial round (OSN-P) and the national round (OSN). **Table 1.1** summarizes them. For 2026, press coverage of the ministry's guide reports that the provincial round for senior high schools ran online from July 6 to 9, 2026 [3]. The national round in informatics was held in Malang from September 14 to 20, 2026, with contest days of five hours (the second day ran from 07:30 to 12:30 WIB) [4].

Table 1.1 The road from a school nomination to the international team, as described by TOKI and the 2026 OSN guide.

Stage	Who takes part	What it asks of you
OSN-K (city/regency)	Students nominated by their schools	Mostly short-answer problems: logic, counting, reading and tracing code
OSN-P (province)	The best of each city or regency	Harder reasoning problems and, in recent years, programming problems
OSN (national)	The best of each province	Two contest days of programming problems with subtasks, five hours each
TOKI training camp	Selected national medallists	Several training and selection stages toward the IOI team

The syllabus follows the syllabus of the International Olympiad in Informatics, with local adjustments. The 2026 OSN syllabus lists ten areas [2]: programming fundamentals; logic and bitwise operations; arithmetic and number theory; counting and probability; recursion, including divide and conquer and backtracking; searching and sorting; problem-solving strategies such as complete search, greedy algorithms and dynamic programming; data structures from arrays and stacks to heaps and trees; graphs and trees, including traversal, shortest paths and spanning trees; and basic geometry. Every chapter of this book sits inside that list.

1.2.2 How the national round is scored

The national round follows the format of the IOI. The published rules of recent national rounds describe two contest days of five hours, with three problems per day [5]. Each problem is worth 100 points and is divided into *subtasks*: groups of test cases with extra restrictions, each worth part of the 100 points. A subtask might say " $N \leq 10$ ", or "all values are equal", or "the tree is a path". You earn a subtask's points only when every test in it passes, and the score that counts for a subtask is the maximum over all your submissions, so a later, worse submission never takes points away. The total is the sum over all problems of both days; ties share a rank, and the time of submission does not matter [5].

The 2026 results page of TOKI shows what this means in practice [6]. With six problems the maximum is 600 points. The highest score in 2026 was 552; the page lists 5 gold, 10 silver and 16 bronze medals and 10 honourable mentions, and the lowest medal score was 197 points, about a third of the maximum. Read that number again: a bronze medal at the national olympiad needed roughly two problems' worth of points spread across six problems. Most of those points were subtasks.

1.2.3 After the national round: TOKI and the IOI

TOKI prepares the best Indonesian secondary-school students to represent the country in international competitions [7]. The medallists of the national round are the pool from which the training camp is formed, and the camp, through several stages, selects the four students who go to the IOI. IOI 2026 was held in Tashkent, Uzbekistan, from August 9 to 16, 2026, with 375 contestants from 92 countries and delegations; its gold, silver and bronze thresholds were about 361, 303 and 229 points out of 600 [8]. All four Indonesian contestants came home with bronze medals [9]. The contest there looks exactly like the national round: two days, three problems a day, subtasks, and a judge that shows your score as you submit.

1.2.4 The contest environment

The national rules published by TOKI describe a contest where C++ is the language, compiled with GCC in the C++17 or C++20 mode, and where you may use blank paper, an offline editor and compiler, and the documentation of the language, but nothing else [5]. There is a limit on the number of submissions per problem, and problems may be *batch* (read input, write output), *interactive* (talk to a grader program) or *output-only*. Details change from year to year, so read the rules page of your year on the TOKI site in the week before the contest, and practise in the same conditions: a plain editor, the compiler in a terminal, and no internet.

KEY INSIGHT: The scoring rule “the maximum over all submissions, per subtask” changes how you should work. A submission can never cost you points. Submit early, submit often, and treat every submission as a way to bank a subtask, not as a final answer.

1.3 Your Workbench in Ten Minutes

Everything in this book is written in C++17 and compiled with `g++ -std=c++17 -O2 -Wall -Wextra`, which is what the national round and SPOJ both accept. The full installation, with screenshots and troubleshooting for every system, is in Chapter 1 of *Competitive Programming with SPOJ* [1]. Here is the short version, with the versions current in September 2026:

- **Windows.** Install MSYS2 and, in its UCRT64 terminal, the package `mingw-w64-ucrt-x86_64-gcc` (GCC 16.2.0 in September 2026) [10]. Add its `bin` folder to the `PATH`.
- **macOS.** Run `xcode-select --install`; the Command Line Tools provide Apple’s Clang, which also answers to `g++`. If you want the real GCC, install it with Homebrew.
- **Linux.** Install `build-essential` (Ubuntu and Debian) or the `gcc-c++` package (Fedora). GCC 16.2, released on August 7, 2026, is the newest release [11].
- **Editor.** Visual Studio Code (version 1.139 in September 2026) [12] with Microsoft’s C/C++ extension; Python 3 is needed only for the small test generators in this book.

The code for this chapter lives in the folder `Code_new/Chapter01`, and every chapter’s folder has the same shape: the solution named after the SPOJ problem, a slow `brute.cpp`, a random-input generator `gen.py`, a script `stress.sh` that compares the two programs, the warm-up programs, sample inputs, and a `README.md` with the commands for Windows, macOS and Linux.

Train in the contest’s conditions. At the national round you will not have your favourite plug-ins, and the machine may not be yours. So, at least once a week, write a whole solution in a plain editor and compile it in a terminal. Know by heart how to compile, how to redirect input from a file (`./prog < in.txt`), and how to compare two files (`cmp` or `diff` on Linux and macOS, `fc` on Windows).

1.4 Reading Bounds Like a Problem Setter

Every problem statement carries a hidden message in its bounds. The setter chose “ $N \leq 100\,000$ ” and not “ $N \leq 1\,000$ ” because they wanted to kill a particular slow solution. Reading the bounds therefore tells you what kind of algorithm is expected, before you have any idea how to solve the problem.

The rule of thumb is simple. A modern judge machine executes somewhere around 10^8 simple C++ operations per second. If your algorithm performs $f(N)$ basic steps on the largest input and the time limit is t seconds, you want

$$f(N_{\max}) \leq 10^8 \cdot t \quad (1.1)$$

with a comfortable margin, because constant factors, cache misses and slow input can easily cost a factor of three. Equation (1.1) turns a bound into a list of candidate complexities, which Table 1.2 spells out.

Table 1.2 Reading the bounds: the largest input tells you which complexities survive a one-second limit.

Largest N	Safe complexity	Typical techniques
$N \leq 20$	$O(2^N \cdot N)$	bitmasks, backtracking, meet in the middle
$N \leq 500$	$O(N^3)$	interval DP, Floyd–Warshall
$N \leq 5\,000$	$O(N^2)$	quadratic DP, all pairs
$N \leq 200\,000$	$O(N \log N)$	sorting, segment trees, Fenwick trees, binary search on the answer
$N \leq 10^7$	$O(N)$	sieves, prefix sums, two pointers, monotonic stacks
$N \leq 10^{18}$	$O(\log N)$ or $O(\sqrt{N})$	fast exponentiation, matrix power, digit DP

Here is the table at work. Suppose a problem gives an array of $N \leq 50\,000$ numbers and $M \leq 50\,000$ questions, each about a range of the array, with a one-second limit. Answering each question by scanning the range costs up to $N \cdot M = 2.5 \times 10^9$ steps: far over the budget. An $O(\log N)$ answer per question costs about $50\,000 \times 16 = 8 \times 10^5$ steps: two thousand times under the budget. The bounds have just told you to look for a data structure that answers a range question in logarithmic time. That is precisely Chapter 2.

The subtasks carry the same message, one level down. A subtask “ $N \leq 1\,000$ ” says: the setter accepts $O(N^2)$ here. A subtask “all queries have $l = 1$ ” says: prefix answers suffice here. Each subtask is a hint, and each hint is points.

1.5 The Contest Day, Minute by Minute

1.5.1 Minutes 0 to 10: map everything

Do not touch the keyboard. Read every problem of the day, all three of them, including the subtasks. For each one, write four lines on paper: the problem as one sentence of mathematics (“given an array and ranges, report the best subarray inside each range”); its bounds and what Table 1.2 says about them; a guess at the family of the solution; and the list of subtasks with their points, marking those you can already solve. These ten minutes feel expensive. They are the cheapest minutes of the day, because they stop you from pouring two hours into the hardest problem while an easy 40 points waits unread on the next page.

1.5.2 Points per minute

With the map in hand, order the work by the expected points per minute. For every subtask s you believe you can solve, estimate its value v_s in points, the minutes c_s it will cost, and the probability p_s that your attempt will be correct. Then its priority is

$$\text{priority}(s) = \frac{p_s \cdot v_s}{c_s} \quad (1.2)$$

and you work in decreasing order of priority. Nobody computes Equation (1.2) to two decimal places in the contest hall; its value is the habit of asking the question. A brute force for a 12-point subtask that takes ten minutes has priority about 1.2 points per minute; a clever full solution worth 60 more points that might take ninety minutes, with a fifty percent chance of being right, has priority about 0.3. Do the brute force first. It also becomes the checker of the full solution later, so its value is higher than its points.

1.5.3 The shape of a good day

Figure 1.1 shows a plan that many strong contestants follow in some form. The first ten minutes map the day. The next hour banks every easy subtask across all three problems. Then comes the main attack on the problem that offers the most points per minute, followed by the second and third problems. The last half hour is frozen: no new ideas, only submitting what is ready, checking that the best version of each program has been submitted, and re-reading the output format.

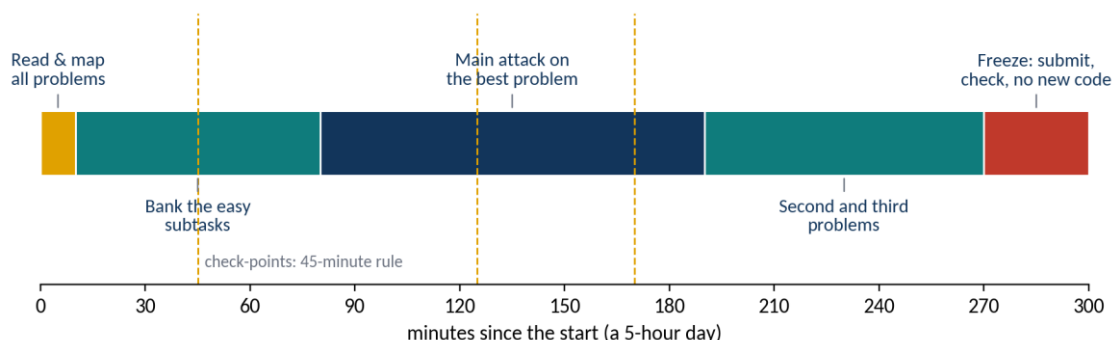


Figure 1.1 A five-hour contest day: map, bank the easy subtasks, attack, finish the other problems, and freeze the last half hour; the dashed lines are check-points for the 45-minute rule.

1.5.4 The 45-minute rule

The most common way to lose a medal is to fall in love with one problem. The rule that protects you is blunt: if you have worked on the same problem for 45 minutes without measurable progress (a new subtask accepted, a proof completed, a bug found), stop. Write down on paper where you are, submit whatever earns points, and move on. You may come back later, and you often will, with a fresh head. The dashed lines in **Figure 1.1** are the moments to ask yourself honestly whether the last 45 minutes produced anything.

1.5.5 Before the day

A champion does not write a segment tree from scratch in the contest hall; the tree flows out of the fingers because it has been written fifty times in training. Keep a personal kit of the structures you use most (fast input, a disjoint-set union, breadth-first search, Dijkstra, a segment tree, a Fenwick tree, modular arithmetic) and practise rewriting each from memory in a few minutes; Halim, Halim and Effendy [13] give a longer checklist of such routines. Keep also an *error*

notebook: after each practice session, write one sentence about the bug that cost you the most time. Read the notebook the evening before the contest. It is a map of your personal traps.

Look after the machine that does the thinking. Sleep properly on the last two nights, eat a real breakfast, bring water, and arrive early enough to sit down calmly. None of this is mysterious, and all of it shows in the fourth hour of a five-hour contest.

1.6 Subtasks: The Mathematics of Medals

Let us make the arithmetic of partial points concrete. Suppose a day has three problems, and consider two contestants.

Ayu solves problem A completely (100 points) and spends the rest of the day fighting problem B without success. Her score is 100.

Bima writes the brute-force subtasks of all three problems in the first ninety minutes: 20 points on A, 15 on B and 12 on C. He then solves a second subtask of A with a greedy idea (30 more points), and in the last two hours finds the intended idea for C's middle subtask (35 more). His score is $20 + 15 + 12 + 30 + 35 = 112$.

Bima never solved a problem completely, and he beat Ayu. Over two days and with national medal thresholds around a third of the maximum, as in 2026 [6], this pattern decides medals. Three habits follow:

- The small- N subtask is almost always a brute force you can write in ten to fifteen minutes. Write it, submit it, bank it.
- Keep every accepted version. Never overwrite working code with an experiment; copy the file first (`A_brute.cpp`, `A_v2.cpp`).
- That same brute force becomes the reference for your fast solution in a stress test (Section 1.8). One piece of code, two uses.

1.7 Warm-Up: Cutting a Line into Words

The case study of this chapter reads English sentences, so the first skill to practise is turning a line of text into words and an end mark. It sounds trivial, and it is the source of a surprising number of wrong answers: double spaces, a word glued to its full stop, a line without any end mark, a Windows carriage return at the end of the line. **Listing 1.1** does it with one pass over the characters.

Listing 1.1 `tokenize.cpp`: split each line into words and report the first end mark.

```
// Warm-up: split each line into words and report its end mark.
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
    string line;
    while (getline(cin, line)) {
        vector<string> words;
        string word;
        char end = '-'; // '-' means: no end mark found
        for (char c : line) {
            if (c == '.' || c == '?' || c == '!' || c == ' ') {
                if (!word.empty()) words.push_back(word);
                word.clear();
                if (c != ' ') { end = c; break; }
            } else {
                word += c;
            }
        }
        cout << "end=" << end << " words=" << words.size() << ":";
        for (const string& w : words) cout << " [" << w << "]";
        cout << "\n";
    }
    return 0;
}
```

The loop keeps the word being built in `word`. A space, or any of the three end marks, *closes* the word: if it is not empty it goes into `words`. An end mark also stops the scan, so anything after the full stop is ignored. On this input:

```
tom  likes big cats.
who eats fish?
bye!
no mark here
```

the program prints:

Look at the first line: two spaces between `tom` and `likes` produce no empty word, because an empty `word` is never pushed. Look at the last line: the word `here` is missing. It was never closed, because no space and no end mark followed it. That is the correct behaviour for our purpose (a line without an end mark is not a sentence), but it is exactly the kind of detail you must decide on purpose rather than discover on the judge.

1.8 The Debug Protocol and the Stress Test

When a solution fails, panic suggests random changes. The protocol replaces panic with a procedure:

1. **Make the failing input small.** Shrink it until the bug is naked, ideally a handful of numbers you can trace by hand.
2. **Try the edges on purpose.** $N = 1$; all values equal; all values negative; the largest values; empty ranges if they are allowed. These few cases catch a large share of wrong answers.
3. **Check the types.** Any product or sum that can pass about 2×10^9 needs `long long`. Write this on your paper at the start of every contest.
4. **Pick one index convention for life.** Either 0-based or 1-based arrays, never a mixture in one program.

5. Stress-test. Compare your fast program with a slow, obviously correct one on many small random inputs.

The fifth step is the most powerful, and every case study in this book uses it. You need three programs: the solution, a *brute force* that solves the same problem in the most direct way you can think of (slow does not matter on small inputs), and a *generator* that prints a random small input from a seed. A short script runs them in a loop and stops at the first input where the two answers differ. **Listing 1.2** is the script used in every chapter.

Listing 1.2 `stress.sh`: compile both programs, then compare them on random inputs until they differ.

```
#!/usr/bin/env bash
# Stress test: compare the solution with the brute force on random inputs.
# Usage: ./stress.sh PROBLEM [ROUNDS]      (e.g. ./stress.sh HOTLINE 500)
set -e
P=${1:?problem name}; N=${2:-500}
g++ -std=c++17 -O2 -Wall -Wextra -o sol "$P.cpp"
g++ -std=c++17 -O2 -Wall -Wextra -o brute brute.cpp
for ((i = 1; i <= N; i++)); do
    python3 gen.py "$i" > in.txt
    ./sol < in.txt > out_sol.txt
    ./brute < in.txt > out_brute.txt
    if ! cmp -s out_sol.txt out_brute.txt; then
        echo "DIFFERENT on seed $i"; cat in.txt; exit 1
    fi
done
echo "all $N tests identical"
```

Two details matter. The generator takes the loop counter as its seed, so a failing input is reproducible: `python3 gen.py 137` prints the same input tomorrow. And the inputs are *small*, so that when the programs differ, the input is short enough to trace by hand. A stress test does not prove a program correct; it finds, in a minute, the kind of mistake that would otherwise cost an hour of staring.

TRAP: A brute force that shares code with the solution also shares its bugs. Write the brute force from the problem statement, in the most naive way, without looking at the fast program.

1.9 HARD Case Study: HOTLINE

1.9.1 The problem in our own words

The SPOJ problem HOTLINE [14] imagines a telephone hotline that talks with a caller. The caller says simple English sentences, one per line. Some are *statements*, ending with a full stop, such as `tom likes fish.` or `I don't eat fish.` or `everybody sees big cats.` Some are *questions*, ending with a question mark, such as `does tom like fish?`, `who likes fish?` or `what does tom do?`. The hotline must remember every statement and answer every question using only what it has heard so far, in precisely the grammar the statement prescribes. If the caller ever says two things that cannot both be true, the hotline gives up: from that moment it answers every question with `I am abroad.` A line ending with an exclamation mark ends the dialogue. The input contains several dialogues; for each one the program prints `Dialogue #k:`, then every question followed by its answer and a blank line, and finally the closing line followed by a blank line.

The difficulty is not speed: a dialogue has at most about a hundred sentences. The difficulty is fidelity. The subjects `I` and `you` take the verb without an `s`; everyone else takes it with an `s`. The caller's `I` is the hotline's `you`, so the answers swap them. `everybody` and `nobody` are subjects that

speak for everyone. Several subjects are joined as A, B and C. A single missing comma is a wrong answer, and SPOJ will not tell you which one.

1.9.2 A sample dialogue

Here is a two-dialogue input we made up to exercise most of the rules:

```
2
I like fish.
tom doesn't eat fish.
ann likes fish.
everybody sees big cats.
do you like fish?
does tom eat fish?
does ann eat fish?
who likes fish?
who sees big cats?
what does tom do?
see you!
nobody eats fish.
tom eats fish.
do I eat fish?
bye!
```

and the output of the solution:

Several rules are visible already. The caller said `I like fish.`, but the question `do you like fish?` asks about the hotline, about which nothing is known: `maybe`. The answer to `who likes fish?` swaps the caller's `I` into `you` and joins two subjects with `and`, using the verb without `s` because the subject is plural. `everybody sees big cats.` answers `who sees big cats?` on its own, and it also appears in the answer to `what does tom do?`. In the second dialogue, `nobody eats fish.` and `tom eats fish.` cannot both be true, so the question that follows is answered `I am abroad`.

1.9.3 Modeling: facts as keys

The first design decision is how to store what the caller said. Every statement, once normalized, is a *fact* with four parts: a subject, a sign (positive or negative), a verb in its base form, and an object (possibly empty, possibly several words). `tom doesn't eat fish.` becomes (`tom`, negative, `eat`, `fish`); `nobody eats fish.` becomes (`nobody`, negative, `eat`, `fish`), because “nobody eats” is a negative statement about everyone. Normalizing the verb means removing the third-person `s`: `likes` becomes `like`. For `I` and `you` the verb is already in base form.

For fast lookups, each fact is also written as a string key `subject|verb|object`, stored in one of two dictionaries, `posMap` for positive facts and `negMap` for negative ones. The vertical bar is a separator that cannot occur in a word, so different facts never produce the same key.

When is a dialogue contradictory? Write a positive fact as P and a negative fact as Q , both about the same verb and object. They contradict each other exactly when

$$P.subject = Q.subject \vee P.subject = \text{everybody} \vee Q.subject = \text{nobody} \quad (1.3)$$

In words: the same person both does and does not; or everybody does while someone does not; or nobody does while someone does. Equation (1.3) is symmetric, so it can be checked *incrementally*: when a new fact arrives, it is enough to compare it with the stored facts of the opposite sign. A new positive fact with key k clashes if `negMap` contains k , or contains the key with

subject **nobody**, or, when the new subject is **everybody**, contains *any* key ending in **|verb|object**. The negative case is the mirror image.

Answering a yes/no question **do X verb object?** is two lookups. If the positive fact about X exists, or the positive fact about **everybody**, the answer is **yes**, ... Otherwise, if the negative fact about X exists, or the negative fact about **nobody**, the answer is **no**, ... Otherwise **maybe**.. The questions **who** and **what** need the facts in the order they were said, so the program also keeps a list **stmts** of distinct facts in order of appearance.

1.9.4 The full solution

Listing 1.3 is the complete solution. Its logic is that of a solution recorded as ACCEPTED on SPOJ, reorganized into small functions so that each rule of the statement has one home.

Listing 1.3 **HOTLINE.cpp**: facts as keys, incremental contradiction checks, and one function per kind of question.

```
// HOTLINE (SPOJ): remember simple English statements, detect contradictions
// and answer do/does, who and what questions. C++17, ACCEPTED Logic.
#include <cstdio>
#include <cstring>
#include <map>
#include <set>
#include <string>
#include <vector>
using namespace std;

static char buf[128];

bool isIorYou(const string& s) { return s == "I" || s == "you"; }
// "Like" for I/you, "Likes" for everyone else
string verbS(const string& v, const string& s) { return isIorYou(s) ? v : v + "s"; }
string dontS(const string& s) { return isIorYou(s) ? "don't" : "doesn't"; }
// the speaker's "I" is the hotline's "you" and vice versa
string swapIY(const string& s) {
    if (s == "I") return "you";
    if (s == "you") return "I";
    return s;
}

string baseVerb(const string& v) { // "Likes" -> "Like"
    if (!v.empty() && v.back() == 's') return v.substr(0, v.size() - 1);
    return v;
}

string joinFrom(const vector<string>& w, size_t from) { // words from..end
    string r;
    for (size_t i = from; i < w.size(); i++) {
        if (!r.empty()) r += " ";
        r += w[i];
    }
    return r;
}

string finish(string s, const string& obj) { // add " obj" (if any) and "."
    if (!obj.empty()) s += " " + obj;
    return s + ".";
}

struct Stmt { string subj; bool neg; string verb, obj; };

struct Dialogue {
    vector<Stmt> stmts; // distinct facts, in order of appearance
    map<string, bool> posMap, negMap; // keys "subject|verb|object"
    bool contradiction = false;
};
```

Listing 1.3 (Continued) *HOTLINE.cpp: facts as keys, incremental contradiction checks, and one function per kind of question.*

```
bool anyKeyEndsWith(const map<string, bool>& m, const string& suffix) {
    for (const auto& kv : m) {
        const string& k = kv.first;
        if (k.size() >= suffix.size() &&
            k.compare(k.size() - suffix.size(), suffix.size(), suffix) == 0)
            return true;
    }
    return false;
}

void addStatement(const vector<string>& words) {
    string subj = words[0], verb, obj;
    bool neg = false;
    if (words[1] == "don't" || words[1] == "doesn't") {
        if (words.size() < 3) return;
        neg = true;
        verb = words[2];
        obj = joinFrom(words, 3);
    } else {
        verb = isIorYou(subj) ? words[1] : baseVerb(words[1]);
        obj = joinFrom(words, 2);
        if (subj == "nobody") neg = true;
    }
    string tail = "|" + verb + "|" + obj, key = subj + tail;
    map<string, bool>& mine = neg ? negMap : posMap;
    if (!mine.count(key)) {
        mine[key] = true;
        stmts.push_back({subj, neg, verb, obj});
    }
    if (contradiction) return;
    if (!neg) { // a "yes" clashes with the same "no" or with "nobody"
        if (negMap.count(key) || negMap.count("nobody" + tail))
            contradiction = true;
        if (subj == "everybody" && anyKeyEndsWith(negMap, tail)) contradiction = true;
    } else { // a "no" clashes with the same "yes" or with "everybody"
        if (posMap.count(key) || posMap.count("everybody" + tail))
            contradiction = true;
        if (subj == "nobody" && anyKeyEndsWith(posMap, tail)) contradiction = true;
    }
}
```

Listing 1.3 (Continued) *HOTLINE.cpp: facts as keys, incremental contradiction checks, and one function per kind of question.*

```

string answerDo(const vector<string>& words) { // do/does X verb obj?
    string qsubj = words[1], qverb = words[2], qobj = joinFrom(words, 3);
    string tail = "|" + qverb + "|" + qobj, who = swapIY(qsubj);
    if (posMap.count(qsubj + tail) || posMap.count("everybody" + tail))
        return finish("yes, " + who + " " + verbS(qverb, who), qobj);
    if (negMap.count(qsubj + tail) || negMap.count("nobody" + tail))
        return finish("no, " + who + " " + dontS(who) + " " + qverb, qobj);
    return "maybe.";
}

// <P50>
// answerWho: List the subjects of positive facts with this verb and object
// (everybody wins; "A, B and C"; else nobody; else "I don't know.").
// answerWhat: List X's distinct facts, own plus everybody/nobody, in order.
// >>> YOUR TURN (Public Edition): write this part yourself, following the hint above.
<<<
// </P50>

string answer(const vector<string>& words) {
    if (contradiction) return "I am abroad.";
    if (words.empty()) return "maybe.";
    if ((words[0] == "do" || words[0] == "does") && words.size() >= 3)
        return answerDo(words);
    if (words[0] == "who" && words.size() >= 2) return answerWho(words);
    if (words[0] == "what" && words.size() >= 4) return answerWhat(words);
    return "maybe.";
}

void run() {
    vector<pair<string, string>> out; // (question, answer)
    while (fgets(buf, sizeof(buf), stdin)) {
        int len = (int)strlen(buf);
        while (len > 0 && (buf[len - 1] == '\n' || buf[len - 1] == '\r'))
            buf[--len] = 0;
        if (len == 0) continue;
        char end = 0;
        vector<string> words;
        string word;
        for (int i = 0; i < len && !end; i++) {
            char c = buf[i];
            if (c == '.' || c == '?' || c == '!' || c == ' ') {
                if (!word.empty()) words.push_back(word);
                word.clear();
                if (c != ' ') end = c;
            } else {
                word += c;
            }
        }
    }
}

```

Listing 1.3 (Continued) *HOTLINE.cpp: facts as keys, incremental contradiction checks, and one function per kind of question.*

```

    }
    if (end == '!') { // the dialogue is over: print everything
        for (auto& qa : out)
            printf("%s\n%s\n", qa.first.c_str(), qa.second.c_str());
        printf("%s\n", buf);
        return;
    }
    if (end == '.' && words.size() >= 2) addStatement(words);
    else if (end == '?') out.push_back({buf, answer(words)});
}
};

int main() {
    int T;
    if (scanf("%d", &T) != 1) return 0;
    for (int d = 1; d <= T; d++) {
        printf("Dialogue #d:\n", d);
        Dialogue dlg;
        dlg.run();
    }
    return 0;
}

```

1.9.5 Line by line

Grammar helpers. `isIorYou`, `verbS` and `dontS` encode the only piece of English grammar the problem needs: `I` and `you` take `like` and `don't`, every other subject takes `likes` and `doesn't`. `swapIY` turns the caller's point of view into the hotline's. `baseVerb` removes one trailing `s`. `joinFrom` glues the remaining words into the object, and `finish` appends the object (when there is one) and the full stop, so that no answer can forget either.

Storing a statement. `addStatement` receives the words of a sentence that ended with a full stop and has at least two words. If the second word is `don't` or `doesn't`, the fact is negative, the verb is the third word and the object starts at the fourth. Otherwise the verb is the second word, normalized unless the subject is `I` or `you`, and a subject `nobody` makes the fact negative. The fact is stored only if its key is new, so repeating a sentence does not repeat it in later answers. Then, unless the dialogue is already contradictory, the incremental check of [Equation \(1.3\)](#) runs. `anyKeyEndsWith` performs the “any subject” part by comparing the end of every key with `|verb|object`.

Yes/no questions. `answerDo` is the two-lookup rule described above. Note that the lookup uses the question's subject as written (`do you ...` looks for facts about `you`), while the answer uses the swapped subject.

Who and what. Between the markers `// <P50>` and `// </P50>` sit the two list-building answers. `answerWho` scans the facts in order. A positive fact about `everybody` settles the question immediately. Otherwise the subjects of the positive facts are collected once each, swapped into the hotline's point of view, and joined with commas and a final `and`; a single subject decides the verb form, several subjects take the plural. If no one is known to do it but `nobody ...` was said, the answer is `nobody ...`; otherwise `I don't know`. `answerWhat` collects, in order, every distinct fact whose subject is the asked person, `everybody` or `nobody`, and writes them as one sentence: the first action follows the subject, the others are separated by commas, and the last is introduced by `, and`.

The dialogue loop. `run` reads lines with `fgets`, strips the line ending (including a Windows `\r`) and skips empty lines. It cuts the line into words exactly as **Listing 1.1** does. A `!` ends the dialogue: the program prints all stored questions and answers, then the closing line. The answers are stored rather than printed at once because the output of each dialogue is printed as one block after its last line has been read. Finally, `main` reads the number of dialogues and runs a fresh `Dialogue` object for each, so no fact leaks from one dialogue into the next.

1.9.6 Traps

- **Mixing points of view.** The caller's `I` is stored as `I`, and the question `do I ...` looks up `I`; only the *answer* says `you`. Swapping at storage time instead gives wrong answers to half the questions.
- **Forgetting that *nobody* is negative.** `nobody eats fish.` has no `doesn't` in it, yet it is a negative fact; missing this breaks both the contradiction check and the *who* answer.
- **Answering after a contradiction.** Once the flag is set, *every* later question in that dialogue answers `I am abroad.`, including questions whose answer is known.
- **Duplicates.** A statement said twice must appear once in the *who* and *what* answers.
- **Output layout.** Each question and its answer are followed by an empty line, and so is the closing line. The judge compares the output exactly.
- **Stale state.** A fact from dialogue 1 must not answer a question in dialogue 2. Creating a new `Dialogue` object per dialogue makes this automatic.

1.9.7 Stress test against a brute force

What is a “brute force” for a reading problem? Not a slower algorithm, but an *independent* implementation of the same rules, written in the most direct way. The file `brute.cpp` in the chapter folder keeps a plain list of facts, rechecks every pair of facts with `Equation` (1.3) after every statement, and answers every question by scanning the list. It shares no code with **Listing 1.3**. The generator **Listing 1.4** builds random dialogues from a tiny vocabulary (four people, three verbs, three objects), so that repeated facts, `everybody`, `nobody` and contradictions all occur often.

Listing 1.4 `gen.py`: random HOTLINE dialogues from a four-person, three-verb vocabulary.

```
#!/usr/bin/env python3
"""Random HOTLINE dialogues from a tiny vocabulary: python3 gen.py SEED"""
import random
import sys

rng = random.Random(int(sys.argv[1]) if len(sys.argv) > 1 else 1)
PEOPLE = ["I", "you", "tom", "ann"]
VERBS = ["like", "eat", "see"]
OBS = ["", "fish", "big cats"]

def verb_for(subj, verb):
    return verb if subj in ("I", "you") else verb + "s"

def statement():
    v, o = rng.choice(VERBS), rng.choice(OBS)
    r = rng.random()
    if r < 0.08:
        words = ["everybody", v + "s"]
    elif r < 0.14:
        words = ["nobody", v + "s"]
    else:
        s = rng.choice(PEOPLE)
        if rng.random() < 0.3:
            words = [s, "don't" if s in ("I", "you") else "doesn't", v]
        else:
            words = [s, verb_for(s, v)]
    return " ".join(words + ([o] if o else [])) + "."

def question():
    v, o = rng.choice(VERBS), rng.choice(OBS)
    tail = [o] if o else []
    kind = rng.randrange(3)
    s = rng.choice(PEOPLE)
    if kind == 0:
        aux = "do" if s in ("I", "you") else "does"
        return " ".join([aux, s, v] + tail) + "?"
    if kind == 1:
        return " ".join(["who", v + "s"] + tail) + "?"
    aux = "do" if s in ("I", "you") else "does"
    return " ".join(["what", aux, s, "do"]) + "?"
```

Listing 1.4 (Continued) *gen.py: random HOTLINE dialogues from a four-person, three-verb vocabulary.*

```
T = rng.randint(1, 3)
print(T)
for _ in range(T):
    for _ in range(rng.randint(1, 12)):
        print(statement() if rng.random() < 0.6 else question())
    print("bye!")
```

Running the stress script for 300 rounds prints:

```
all 300 tests identical
```

The two programs also agree with the original ACCEPTED code on the same inputs, which confirms that the reorganization changed nothing. The final check is, of course, a submission to SPOJ itself.

1.10 Contest-Day Tactics

- **Map for ten minutes.** Read all problems and all subtasks before writing code; write one line of mathematics per problem.
- **Bank cheap subtasks early.** The brute-force subtask of every problem is usually worth ten to twenty points and costs ten to fifteen minutes.
- **Obey the 45-minute rule.** No measurable progress in 45 minutes means you move on.
- **Keep every accepted version.** Copy before you experiment.
- **Stress-test before you trust.** A brute force plus a generator plus [Listing 1.2](#) catches the mistakes that samples never show.
- **Read the output format twice.** Once at the start, once in the frozen last half hour. **HOTLINE** shows how a single comma can cost a whole problem.
- **Freeze the last thirty minutes.** Submit, check, and resist the brilliant new idea.

When panic arrives, and in a five-hour contest it will, write two lists on paper: what is certainly correct (tested, accepted) and what is not yet. Panic comes from mixing the two. Separating them in ink turns the feeling back into a plan.

1.11 Summary

- OSN informatics has three stages (city or regency, province, national); the national round has two five-hour days of three IOI-style problems with subtasks, and in 2026 a medal needed roughly a third of the 600 available points.
- The best submission counts per subtask, so submitting never costs points and partial solutions are worth a great deal.
- The bounds of a problem, read through [Equation \(1.1\)](#) and [Table 1.2](#), tell you which complexities can pass before you know the solution.
- Order your work by expected points per minute, [Equation \(1.2\)](#), and protect yourself with the 45-minute rule and a frozen final half hour.
- A stress test compares a fast solution with an independent brute force on many small random inputs and finds most bugs in minutes.
- **HOTLINE** is solved by modeling each sentence as a signed fact, storing facts as keys, checking contradictions incrementally with [Equation \(1.3\)](#), and giving each kind of question its own small function.

1.12 Exercises

1. A problem has $N \leq 3\,000$ and a two-second limit. Using [Equation \(1.1\)](#), which of $O(N^3)$, $O(N^2 \log N)$ and $O(N^2)$ are safe?
2. A day offers these subtasks: A1 worth 15 points (10 minutes, you are sure), A2 worth 40 (80 minutes, fifty-fifty), B1 worth 20 (25 minutes, fairly sure, say 0.8), C1 worth 10 (20 minutes, sure). Order them by [Equation \(1.2\)](#).
3. Run [Listing 1.1](#) by hand on the line `who sees?big cats.` How many words does it report, and which end mark?
4. In **HOTLINE**, the dialogue `everybody likes fish. / ann doesn't like fish. / does tom like fish? / bye!` is given. What does the program print, and why?

5. What does the solution answer to `who eats fish?` after `tom eats fish.`, `I eat fish.`, `ann eats fish.` and `tom eats fish.`? Write the answer exactly.
6. Change the generator so that it never produces contradictions. Why would that make the stress test weaker?
7. Explain why a submission that scores less than an earlier one never lowers your total at the national round.

1.13 Solutions to the Exercises

1. The budget is $10^8 \cdot 2 = 2 \times 10^8$. $N^3 = 2.7 \times 10^{10}$: far too slow. $N^2 \log N \approx 9 \times 10^6 \times 12 \approx 1.1 \times 10^8$: inside the budget, but with little margin for a large constant. $N^2 = 9 \times 10^6$: very safe.
2. Priorities: A1 $1 \cdot 15/10 = 1.5$; C1 $1 \cdot 10/20 = 0.5$; B1 $0.8 \cdot 20/25 = 0.64$; A2 $0.5 \cdot 40/80 = 0.25$. Order: A1, B1, C1, A2.
3. The characters `who`, two spaces, `sees`, then `?`. The spaces close `who` (the second space finds an empty `word` and adds nothing); the question mark closes `sees` and stops the scan. Output: `end=? words=2: [who] [sees]`. The text after `?` is ignored.
4. `ann doesn't like fish.` contradicts `everybody likes fish.` by the second case of Equation (1.3), so the flag is set before the question arrives. The program prints `Dialogue #1:`, then `does tom like fish?`, then `I am abroad.`, an empty line, `bye!` and an empty line.
5. Distinct positive subjects in order of appearance: `tom`, `I`, `ann` (the repeated `tom` is ignored). Swapped: `tom`, `you`, `ann`. Three subjects take the plural verb: `tom`, `you` and `ann eat fish.`
6. Contradictions trigger the `I am abroad.` rule and the whole contradiction-checking code, including `anyKeyEndsWith`. A generator without them would never test those lines, so a bug there would pass every round of the stress test.
7. Because the score of each subtask is the maximum over all submissions [5]. A worse submission only adds a smaller number to a maximum that already contains the larger one.

References

- [1] *Competitive Programming with SPOJ: From Reading the Problem to Accepted, in C++ and Python*. Profitina, 2026.
- [2] Tim Olimpiade Komputer Indonesia, "Silabus OSN 2026 — OSN Bidang Informatika." [Online]. Available: <https://osn.toki.id/silabus> (accessed Sep. 2026).
- [3] Suara Pecari, "Panduan Lengkap OSN 2026: Jadwal, Aturan, dan Komponen," Jun. 2026. [Online]. Available: <https://suarapecari.com/pendidikan/sp-2026060535876/panduan-lengkap-osn-2026-jadwal-aturan-dan-komponen/> (accessed Sep. 2026).
- [4] Tim Olimpiade Komputer Indonesia, "OSN 2026 Bidang Informatika, Malang, 14–20 September 2026 — Peringkat." [Online]. Available: <https://peringkat.toki.id/osn-2026/> (accessed Sep. 2026).
- [5] Tim Olimpiade Komputer Indonesia, "Peraturan — OSN Bidang Informatika." [Online]. Available: <https://osn.toki.id/OSN2022/peraturan> (accessed Sep. 2026).
- [6] Tim Olimpiade Komputer Indonesia, "Olimpiade Sains Nasional 2026 Bidang Informatika — Hasil." [Online]. Available: <https://osn.toki.id/OSN2026/hasil> (accessed Sep. 2026).
- [7] Tim Olimpiade Komputer Indonesia, "About." [Online]. Available: <https://toki.id/about/> (accessed Sep. 2026).
- [8] International Olympiad in Informatics, "IOI 2026 — Tashkent, Uzbekistan," IOI Statistics. [Online]. Available: <https://stats.ioinformatics.org/olympiads/2026> (accessed Sep. 2026).

- [9] Kaldera News, “Membanggakan! 4 Pelajar Indonesia Bawa Pulang Medali dari IOI 2026,” Aug. 2026. [Online]. Available: <https://www.kalderanews.com/2026/08/membanggakan-4-pelajar-indonesia-bawa-pulang-medali-dari-ioi-2026/> (accessed Sep. 2026).
- [10] MSYS2 Developers, “Package: mingw-w64-ucrt-x86_64-gcc,” version 16.2.0. [Online]. Available: https://packages.msys2.org/packages/mingw-w64-ucrt-x86_64-gcc (accessed Sep. 2026).
- [11] GNU Project, “GCC, the GNU Compiler Collection,” release list with GCC 16.2 (Aug. 7, 2026). [Online]. Available: <https://gcc.gnu.org/> (accessed Sep. 2026).
- [12] Microsoft, “Visual Studio Code Updates, version 1.139,” Sep. 2026. [Online]. Available: <https://code.visualstudio.com/updates> (accessed Sep. 2026).
- [13] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Lulu Press, 2020.
- [14] Sphere Online Judge, “HOTLINE — Hotline.” [Online]. Available: <https://www.spoj.com/problems/HOTLINE/> (accessed Sep. 2026).

Chapter 2

Segment Trees and Mergeable Answers — GSS1

Almost every national olympiad contains a problem that asks the same kind of question again and again about a *range* of an array: the sum of positions l to r , the smallest value between them, or something stranger. The *segment tree* is the standard tool for all of them, and the real skill it demands is not the tree itself, which is short, but the design of what each node stores. By the end of this chapter you will be able to build a segment tree in linear time, answer a range question in logarithmic time and prove why, and, most importantly, design a node for an unusual question by asking what information is missing when two neighbouring answers are glued together. The case study is the SPOJ classic GSS1, the best-sum subarray inside a range, whose four-field node is one of the most useful small designs in competitive programming.

2.1 A Cooperative's Ledger

A fruit-growers' cooperative in the hills keeps a ledger of its daily net income: the money from the day's sales minus the day's costs. Some days are good (a truck of mangosteen sold in the city), some are bad (rain, a broken pump, fruit that spoiled on the way). The treasurer is often asked the same question in different forms. "Between the 3rd and the 20th of March, what was our best stretch of consecutive days?" "In the durian season, from day 150 to day 210, which run of days earned the most?" The answer is always the largest sum of a block of consecutive days inside the period.

For one period, the treasurer can simply try every block: every starting day and every ending day. For a ledger of a few thousand days and a handful of questions, that is fine. But the cooperative wants a small app where members can ask any period they like, and the ledger grows every day. With N days and M questions, trying every block costs up to N^2 steps per question, and even the smarter one-pass method of Section 2.3 costs N steps per question, $N \cdot M$ in total. With fifty thousand days and fifty thousand questions, that is 2.5×10^9 steps, far beyond one second. We need a structure that has done most of the work in advance and can answer each question in about $\log_2 N$ steps.

That is exactly the SPOJ problem GSS1 [1], and the tool is the segment tree.

2.2 The Idea in Plain Words

Split the array into two halves. Split each half into two halves again, and so on, until every piece has a single element. The pieces form a binary tree: the root is the whole array, its two children are the halves, their children the quarters, and the leaves are single elements. Now store, in every piece, the *answer* to our question for that piece.

Two facts make this useful. First, the answer for a piece can be computed from the answers of its two halves, provided the answers carry the right information; we call that computation **combine**. Second, any range $[l, r]$ can be cut into a small number of pieces of the tree, never more than

about $2\log_2 N$, so the answer for the range is obtained by combining a few stored answers. Building the tree costs one **combine** per piece, and there are fewer than $2N$ pieces. Answering a question costs a few dozen **combines**. The whole difficulty of a hard problem moves into one question: what must a piece store so that **combine** is possible?

2.3 Warm-Up Problems

2.3.1 Range sums without updates

Before building a tree, remember the simplest range tool. If the question is only “the sum of $a_l, \dots, a_r$ ” and the array never changes, prefix sums answer each question in constant time. Define

$$p_0 = 0, \quad p_i = p_{i-1} + a_i \quad (1 \leq i \leq N) \quad (2.1)$$

so that p_i is the sum of the first i elements. Then the sum of $a_l, \dots, a_r$ is $p_r - p_{l-1}$: the first r elements minus the first $l - 1$. **Listing 2.1** implements Equation (2.1).

Listing 2.1 `prefix_sum.cpp`: static range sums in constant time per question.

```
// Warm-up: static range sums with a prefix array, O(1) per question.
#include <cstdio>
#include <vector>
using namespace std;

int main() {
    int n, m;
    if (scanf("%d", &n) != 1) return 0;
    vector<long long> p(n + 1, 0); // p[i] = a[1] + ... + a[i], p[0] = 0
    for (int i = 1; i <= n; i++) {
        long long v;
        if (scanf("%lld", &v) != 1) return 0;
        p[i] = p[i - 1] + v;
    }
    if (scanf("%d", &m) != 1) return 0;
    while (m--) {
        int l, r;
        if (scanf("%d %d", &l, &r) != 2) return 0;
        printf("%lld\n", p[r] - p[l - 1]);
    }
    return 0;
}
```

On the array 3, -4, 2, 5, -1, 2 with the questions [1,6], [2,3] and [1,2]:

```
6
3 -4 2 5 -1 2
3
1 6
2 3
1 2
```

Prefix sums are unbeatable when they apply, and a segment tree that answers static sums is wasted effort. Remember this when a problem looks like a segment tree: first ask whether a prefix trick is enough. It stops being enough in two situations: when the array changes (Chapter 5), and when the question is not a sum that can be “subtracted”. The best block of consecutive days is such a question: there is no way to subtract the best block of the first $l - 1$ days from the best block of the first r days.

2.3.2 The best block of the whole array

For the whole array, the best block has a famous one-pass solution usually credited to Kadane and popularized by Bentley [2]. Walk from left to right and keep e_i , the best sum of a block that *ends exactly at position i* . Such a block either is a_i alone, or extends the best block ending at $i - 1$:

$$e_1 = a_1, \quad e_i = \max(a_i, e_{i-1} + a_i) \quad (i \geq 2) \quad (2.2)$$

The answer is the largest e_i . **Listing 2.2** implements Equation (2.2) with one variable for e and one for the best.

Listing 2.2 `kadane.cpp`: the best non-empty block of the whole array in one pass.

```
// Warm-up: Kadane's algorithm, best non-empty subarray of the whole array.
#include <algorithm>
#include <cstdio>
using namespace std;

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    long long endHere = 0, best = 0;
    for (int i = 0; i < n; i++) {
        long long v;
        if (scanf("%lld", &v) != 1) return 0;
        // best piece ending at i: extend the previous one, or start again at i
        endHere = (i == 0) ? v : max(v, endHere + v);
        best = (i == 0) ? endHere : max(best, endHere);
    }
    printf("%lld\n", best);
    return 0;
}
```

On 3, -4, 2, 5, -1, 2 it finds 8 (the block 2, 5, -1, 2), and on an array of negative numbers -3, -1, -7, -2 it finds -1, the single least negative value:

That second output matters. The problem asks for a *non-empty* block, so when every number is negative the answer is negative. A common “simplification”, starting the best at 0, would print 0 there, which is the sum of the empty block and a wrong answer. Kadane’s method answers one range in $O(N)$; the rest of the chapter answers any range in $O(\log N)$.

2.4 The Segment Tree

2.4.1 Shape and numbering

Number the nodes as in a binary heap: the root is node 1 and covers $[1, N]$; a node v covering $[lo, hi]$ with $lo < hi$ has the middle $mid = \lfloor (lo + hi) / 2 \rfloor$, a left child $2v$ covering $[lo, mid]$ and a right child $2v + 1$ covering $[mid + 1, hi]$. A node with $lo = hi$ is a leaf. No pointers are stored; the children of v are found by arithmetic.

Figure 2.1 shows the tree for the eight-element array 2, -5, 4, -1, 3, -6, 5, 1. Each box shows its range and the four numbers our node will carry (Section 2.5); ignore them for now and look at the shape.

each box: range, then sum, pre, suf, ans

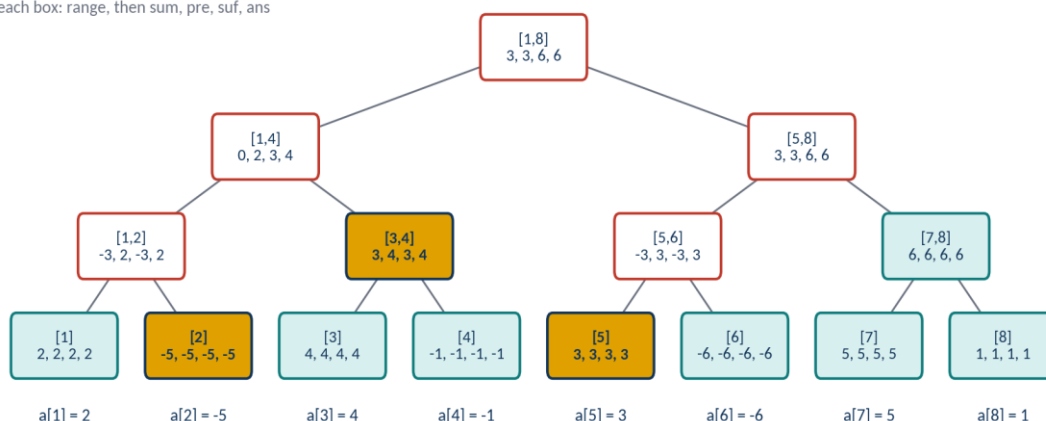


Figure 2.1 The segment tree of 2, -5, 4, -1, 3, -6, 5, 1; the query [2,5] visits the red nodes and combines the three amber nodes [2], [3,4] and [5].

How large must the array of nodes be? The depth of the tree is $\lceil \log_2 N \rceil$, and with heap numbering the largest index used is below $2^{\lceil \log_2 N \rceil + 1}$. Since $2^{\lceil \log_2 N \rceil} < 2N$,

$$\text{largest index} < 2 \cdot 2^{\lceil \log_2 N \rceil} < 4N \quad (2.3)$$

which is why segment-tree code declares `tree[4 * N]`. The bound in Equation (2.3) is not a superstition; allocating only $2N$ nodes works when N is a power of two and crashes for other values, which is one of the most common runtime errors in contests.

2.4.2 Build in linear time

`build(v, lo, hi)` fills node v : a leaf gets the value of its element; any other node first builds its children and then combines them. Every node is visited once, and there are fewer than $2N$ real nodes (a tree with N leaves in which every internal node has two children has exactly $N - 1$ internal nodes), so building costs $O(N)$ `combine` calls.

2.4.3 Query in logarithmic time

`query(v, lo, hi, ql, qr)` returns the answer for the part of $[ql, qr]$ inside node v 's range, which the caller guarantees is non-empty. Three cases:

- If $[lo, hi]$ lies completely inside $[ql, qr]$, the stored node is the answer.
- If the query lies completely in one child ($qr \leq mid$ or $ql > mid$), ask only that child.
- Otherwise the query crosses the middle: ask both children and `combine` the two partial answers, left first.

Why is this fast? Look at any level of the tree. The nodes of that level that the query *visits without being fully covered* must contain an end of the query range, ql or qr , inside them, because a node that contains neither end is either fully inside or fully outside the range. So at most two such nodes exist per level, and each of them makes at most two calls to the level below. Hence at most four nodes are visited per level, and the number of visited nodes satisfies

$$\text{visited}(N) \leq 4\lceil \log_2 N \rceil + 1 = O(\log N) \quad (2.4)$$

The amber nodes in **Figure 2.1** are the *canonical* pieces of $[2,5]$: $[2]$, $[3,4]$ and $[5]$. The red nodes are those visited on the way. Even for $N = 50\,000$, [Equation \(2.4\)](#) promises fewer than seventy visited nodes.

KEY INSIGHT: Nothing in the build or the query knows what the question is. Everything problem-specific lives in two functions: `makeNode` (the answer for one element) and `combine` (the answer for two neighbours). The segment tree is a machine; the node is the program you feed it.

2.5 Designing the Node

2.5.1 The closure recipe

Hard problems never ask for a plain range sum. They ask something unusual, and the job is to find a node that makes the unusual answer *mergeable*. The recipe is:

1. Write down what is asked for a range.
 2. Try to compute it for the union of two neighbouring ranges L and R from their answers alone.
Ask: what information is *missing*?
 3. Add the missing information to the node.
 4. Check that the new fields can themselves be merged. If not, go back to step 2 for them.
- Stop when nothing is missing. The node is then *closed* under combination.

2.5.2 Four fields for the best block

Apply the recipe to the best-sum block. Step 1: store *ans*, the best sum of a non-empty block inside the range. Step 2: the best block of $L \cup R$ is either inside L , or inside R , or it *crosses* the boundary. The first two are $L.ans$ and $R.ans$. The crossing block is a block ending at the right edge of L followed by a block starting at the left edge of R , and its best value is the best *suffix* of L plus the best *prefix* of R . Neither is stored yet. Step 3: add *suf* (best non-empty block touching the right edge) and *pre* (best non-empty block touching the left edge). Step 4: can *pre* be merged? The best prefix of $L \cup R$ either stays inside L ($L.pre$) or takes *all of* L and continues into R ($L.sum + R.pre$). That needs the total *sum*. The sum merges by adding. Nothing is missing any more, and the four fields are:

$$sum = L.sum + R.sum \tag{2.5}$$

$$pre = \max(L.pre, L.sum + R.pre) \tag{2.6}$$

$$suf = \max(R.suf, R.sum + L.suf) \tag{2.7}$$

$$ans = \max(L.ans, R.ans, L.suf + R.pre) \tag{2.8}$$

A leaf holding value x has all four fields equal to x : the only non-empty block of one element is the element itself.

2.5.3 Why the formulas are right

Take [Equation \(2.6\)](#). A non-empty prefix of $L \cup R$ either ends inside L , in which case it is a non-empty prefix of L and at most $L.pre$ (and $L.pre$ itself is such a prefix), or it ends inside R , in which

case it is all of L followed by a non-empty prefix of R , worth at most $L.sum + R.pre$ (and that value is achieved). The maximum of the two cases is therefore exactly the best prefix. Equation (2.7) is the mirror image. For Equation (2.8), every non-empty block of $L \cup R$ lies in L , lies in R , or crosses; a crossing block is a non-empty suffix of L glued to a non-empty prefix of R , and the best such pair is $L.suf + R.pre$ because the two parts can be chosen independently.

One more property is needed: the query in Section 2.4.3 combines partial answers in a particular grouping, for example $(A \oplus B) \oplus C$ where $\oplus$ is `combine`, while the build may group the same pieces as $A \oplus (B \oplus C)$. Both must give the same result. They do, because both are the true four-field description of the concatenated range, and the proof above shows `combine` always produces the true description from true descriptions. In algebraic terms, `combine` is *associative*, but it is not commutative: L must come first.

2.5.4 Working the example by hand

Let us answer the query $[2,5]$ on the array of **Figure 2.1**, where each node is written as (sum, pre, suf, ans) . The three canonical pieces are $[2] = (-5, -5, -5, -5)$, $[3,4] = (3, 4, 3, 4)$ and $[5] = (3, 3, 3, 3)$. First combine $[2]$ with $[3,4]$:

- $sum = -5 + 3 = -2$;
- $pre = \max(-5, -5 + 4) = -1$;
- $suf = \max(3, 3 + (-5)) = 3$;
- $ans = \max(-5, 4, -5 + 4) = 4$.

So $[2,4] = (-2, -1, 3, 4)$. Then combine it with $[5]$:

- $sum = -2 + 3 = 1$;
- $pre = \max(-1, -2 + 3) = 1$;
- $suf = \max(3, 3 + 3) = 6$;
- $ans = \max(4, 3, 3 + 3) = 6$.

The best block inside $[2,5]$ is worth 6: it is $4, -1, 3$, which is exactly the crossing case, the best suffix of $[2,4]$ glued to the best prefix of $[5]$.

2.5.5 The same recipe, other questions

The recipe is worth practising on a few more questions, because the national round rarely asks the same one twice. The idea of storing extra information in the nodes of a tree so that a new question becomes answerable is old and general; Cormen et al. call it *augmenting a data structure* and develop it for balanced search trees [3]. Here are three short exercises in the same spirit, each with the node it leads to.

Minimum and how many times it occurs. Step 1 stores mn . Step 2: the minimum of $L \cup R$ is $\min(L.mn, R.mn)$, nothing missing. For the count, the new count is $L.cnt$ if L 's minimum is strictly smaller, $R.cnt$ if R 's is, and $L.cnt + R.cnt$ on a tie. Two fields, closed.

Is the range sorted? Step 1 stores a flag *ok*. Step 2: $L \cup R$ is sorted when both halves are and the last value of L is at most the first value of R . The edge values are missing, so step 3 adds *first* and *last*, which merge trivially ($L.first$ and $R.last$). Three fields, closed.

Number of "rising" adjacent pairs. A pair (a_i, a_{i+1}) with $a_i < a_{i+1}$ inside $L \cup R$ is inside L , inside R , or is the single pair across the boundary. So the node stores the count plus, again, *first* and *last*, and the merge adds one when $L.last < R.first$.

Notice the pattern that appears in all four designs, GSS1 included: the answer for a range needs information about the *edges* of the range, because that is where two neighbours touch. When you meet a new range question in a contest, the first thing to ask is: “what happens exactly at the boundary between two halves?” The fields that describe the edges are usually the missing ones.

A last warning about the recipe: closure can fail. Some questions, such as “the number of distinct values in $[l, r]$ ”, cannot be closed with a small node, because the two halves may share values in ways that only a full list of values could reveal. When the recipe keeps adding fields without end, stop and look for a different technique; for distinct values, offline processing with a Fenwick tree or the persistent trees of Chapter 3 are the usual answers.

2.6 HARD Case Study: GSS1

2.6.1 The problem in our own words

You are given a sequence of N integers, $a_1, \dots, a_N$, with $N \leq 50\,000$ and every value between $-15\,007$ and $15\,007$. Then M questions follow, each a pair $x \leq y$. For each question, print the largest sum $a_i + a_{i+1} + \dots + a_j$ over all i, j with $x \leq i \leq j \leq y$. The block must be non-empty, so negative answers are possible. The input gives N , then the N values, then M , then the M pairs.

2.6.2 Sample input and output

The sample of the statement has the array $-1, 2, 3$ and one question, $[1, 2]$, whose answer is 2 (the block containing only the 2):

```
3
-1 2 3
1
1 2
```

A second sample of our own uses the array of **Figure 2.1**:

```
8
2 -5 4 -1 3 -6 5 1
4
1 8
2 5
5 7
6 6
```

The second answer is the 6 we computed by hand in Section 2.5.4; the last answer, -6 , is a range of one negative element.

2.6.3 Design

Everything is in place. The bounds ($N, M \leq 50\,000$) ask for $O(\log N)$ per question (Table 1.2 in Chapter 1). The node is the four-field node of Section 2.5. The tree is built once in $O(N)$; each question calls `query` and prints the `ans` field of the node it returns. The one subtle design choice is that `query` returns a whole *node*, not a number: partial answers from the two children are merged with the same `combine` that built the tree, which is exactly what the crossing case needs.

What sizes do the values reach? A sum of up to 50 000 values of absolute value at most 15 007 is at most about 7.5×10^8 in absolute value, which fits in a 32-bit `int`, but only by a factor of three.

The solution uses `long long` in the node, which costs nothing here and removes the question entirely.

2.6.4 The full solution

Listing 2.3 `GSS1.cpp`: the four-field segment tree; build in $O(N)$, each question in $O(\log N)$.

```
// GSS1 (SPOJ): maximum-sum non-empty subarray inside [x, y], static array.
// Segment tree whose nodes carry (sum, pre, suf, ans). C++17.
#include <algorithm>
#include <cstdio>
using namespace std;

const int MX = 50001;
struct Node {
    long long sum; // total of the range
    long long pre; // best sum of a non-empty piece touching the left edge
    long long suf; // best sum of a non-empty piece touching the right edge
    long long ans; // best sum of a non-empty piece anywhere in the range
};
Node tree[4 * MX];
int a[MX];

Node makeNode(long long v) { // a range of one element: every field is v
    Node n;
    n.sum = n.pre = n.suf = n.ans = v;
    return n;
}

Node combine(const Node& l, const Node& r); // defined below

void build(int v, int lo, int hi) {
    if (lo == hi) {
        tree[v] = makeNode(a[lo]);
        return;
    }
    int mid = (lo + hi) / 2;
    build(2 * v, lo, mid);
    build(2 * v + 1, mid + 1, hi);
    tree[v] = combine(tree[2 * v], tree[2 * v + 1]);
}

// <P50>
// combine: glue two neighboring ranges (left l, right r) into one node.
// query: return the node of [ql, qr] inside [lo, hi]; ask only the children
// that overlap, and combine the two partial nodes when both do.
// >>> YOUR TURN (Public Edition): write this part yourself, following the hint above. <<<
// </P50>
```

Listing 2.3 (Continued) *GSS1.cpp: the four-field segment tree; build in $O(N)$, each question in $O(\log N)$.*

```
int main() {
    int n, m;
    if (scanf("%d", &n) != 1) return 0;
    for (int i = 1; i <= n; i++)
        if (scanf("%d", &a[i]) != 1) return 0;
    build(1, 1, n);
    if (scanf("%d", &m) != 1) return 0;
    while (m--) {
        int x, y;
        if (scanf("%d %d", &x, &y) != 2) return 0;
        printf("%lld\n", query(1, 1, n, x, y).ans);
    }
    return 0;
}
```

2.6.5 Line by line

The node. `struct Node` holds the four fields of Equation (2.5) to Equation (2.8), with a comment for each. `tree` has `4 * MX` entries, following Equation (2.3).

The leaf. `makeNode(v)` returns a node whose four fields all equal `v`. It takes a `long long` so that the fields are computed in 64 bits from the start.

Build. `build` is declared after a forward declaration of `combine`, because in the file `combine` is defined further down. A leaf copies `a[lo]` through `makeNode`; any other node builds both children and stores their combination. The call `build(1, 1, n)` in `main` fills the whole tree.

Combine. Between the markers `// <P50>` and `// </P50>` sits the heart of the chapter. `combine` is a literal transcription of Equation (2.5), Equation (2.6), Equation (2.7) and Equation (2.8). The parameters are `const Node&`, which avoids copying 32 bytes per call; the order matters, `l` being the left neighbour.

Query. `query` implements the three cases of Section 2.4.3. A fully covered node returns its stored value. A query entirely in the left half ($qr \leq mid$) or entirely in the right half ($ql > mid$) descends into one child only. The crossing case asks both children and combines the two partial nodes, left before right. Because the caller only descends into children that overlap the query, `query` is never called on an empty intersection, and no “neutral node” is needed. That matters: there is no natural neutral node for a non-empty maximum, and inventing one (for example with very negative values) is a source of overflow bugs.

Main. `main` reads the input with `scanf`, checking every return value so that the program stops cleanly on a truncated input, builds the tree, and prints `query(1, 1, n, x, y).ans` for each question.

2.6.6 Traps

- **Starting from zero.** Writing `pre = max(0, ...)` or initializing a best with 0 computes the best *possibly empty* block. With all-negative ranges it prints 0 instead of the largest element. The quickest test is the array `-1, 2` with the query `[1,2]`: the answer must be 2, and with the array `-3` alone it must be `-3`.
- **Too small a tree.** `tree[2 * N]` fails for most N ; use $4N$ (Equation (2.3)).
- **Combining in the wrong order.** `combine(right, left)` gives wrong *pre* and *suf* and hence wrong crossing answers, yet it often passes samples.

- **Returning a number from `query`.** A query that returns only `ans` cannot handle the crossing case; it must return the whole node.
- **Using `x` and `y` as 0-based.** The problem is 1-based, and so is the tree.

2.6.7 Stress test

The brute force of **Listing 2.4** tries every start i and extends the sum to the right, keeping the best; it is $O(N^2)$ per question and obviously correct.

Listing 2.4 `brute.cpp`: every block of every range, $O(N^2)$ per question.

```
// Brute force for GSS1: try every start i and extend the sum to the right.
#include <algorithm>
#include <climits>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n, m;
    cin >> n;
    vector<long long> a(n + 1);
    for (int i = 1; i <= n; i++) cin >> a[i];
    cin >> m;
    while (m--) {
        int x, y;
        cin >> x >> y;
        long long best = LLONG_MIN;
        for (int i = x; i <= y; i++) {
            long long s = 0;
            for (int j = i; j <= y; j++) best = max(best, s += a[j]);
        }
        cout << best << "\n";
    }
    return 0;
}
```

The generator **Listing 2.5** makes arrays of up to 12 elements whose values are drawn from three scales: tiny (± 3 , many ties and zeros), small (± 10) and the full $\pm 15\,007$.

Listing 2.5 `gen.py`: random GSS1 inputs at three value scales.

```
#!/usr/bin/env python3
"""Random GSS1 input: python3 gen.py SEED"""
import random
import sys

rng = random.Random(int(sys.argv[1]) if len(sys.argv) > 1 else 1)
n = rng.randint(1, 12)
lim = rng.choice([3, 10, 15007])
print(n)
print(*[rng.randint(-lim, lim) for _ in range(n)])
m = rng.randint(1, 10)
print(m)
for _ in range(m):
    x = rng.randint(1, n)
    print(x, rng.randint(x, n))
```

The script `stress.sh` from Chapter 1 (Listing 1.2; the same file is in every chapter folder) runs both programs:

```
all 500 tests identical
```

On the largest input the problem allows, $N = M = 50\,000$ with random values and ranges, the solution takes about 0.03 seconds on our machine, which agrees with Equation (2.4): roughly $50\,000 \times 70$ node visits.

2.7 When Another Tool Is Better

A segment tree is general, but it is not always the best choice. If the array is static and the question is a sum or a count, prefix sums (Equation (2.1)) answer in $O(1)$ with a few lines of code. If the array is static and the question is a minimum or maximum, a *sparse table* answers in $O(1)$ after $O(N \log N)$ preprocessing, because overlapping two pieces does not change a minimum. If the question is “a sum with point updates”, a Fenwick tree is shorter and faster. The segment tree wins when the operation is unusual (like GSS1’s four fields), when updates arrive (Chapter 5), or when both happen. Choosing the smallest tool that works is part of the skill; the textbook treatments of these alternatives in Laaksonen [4] and Halim et al. [5] are worth reading side by side.

2.8 Contest-Day Tactics

- **Keep a tested skeleton.** The functions `build` and `query` of Listing 2.3 are the same for almost every static segment-tree problem. Write them from memory in training until they take five minutes, and change only `Node`, `makeNode` and `combine` in the contest.
- **Design on paper first.** Run the closure recipe of Section 2.5.1 on paper, and check `combine` on two tiny ranges by hand before typing.
- **Bank the $O(N^2)$ subtask.** A subtask with small N is solved by Listing 2.4. Submit it first; it then serves as your stress-test reference.
- **Test the all-negative case.** It is the classic silent failure of maximum-sum problems.

2.9 Summary

- A segment tree stores an answer in every node of a halving tree; the build costs $O(N)$ and every range question costs $O(\log N)$ by Equation (2.4), with an array of $4N$ nodes by Equation (2.3).
- All problem-specific logic lives in `makeNode` and `combine`; `combine` must be associative, and it is often not commutative.
- The closure recipe designs a node: add whatever information is missing when two neighbours are glued, until nothing is missing.
- The best-sum block needs four fields, `sum`, `pre`, `suf` and `ans`, merged by Equation (2.5) to Equation (2.8).
- GSS1 is that node in a static segment tree whose `query` returns whole nodes, so that partial answers combine correctly.
- Prefix sums, sparse tables and Fenwick trees beat the segment tree when their narrower conditions hold.

2.10 Exercises

1. Build the node `[5,8]` of Figure 2.1 from its children `[5,6] = (-3,3,-3,3)` and `[7,8] = (6,6,6,6)` and check it against the figure.

2. For $N = 5$, list the ranges of all nodes of the tree and the largest node index used. Is it below $4N$?
3. Which nodes does `query(1, 1, 8, 3, 8)` return without descending, on the tree of **Figure 2.1**?
4. Design a node for “the length of the longest run of equal adjacent values inside $[l, r]$ ”. Which fields are needed?
5. A friend writes `res.ans = max(l.ans, r.ans)` and forgets the crossing term. Give the smallest input on which the program is wrong.
6. Explain why `combine(query(left), query(right))` would be wrong if `query` were allowed to be called on a node that does not overlap the query.

2.11 Solutions to the Exercises

1. $sum = -3 + 6 = 3$; $pre = \max(3, -3 + 6) = 3$; $suf = \max(6, 6 + (-3)) = 6$; $ans = \max(3, 6, -3 + 6) = 6$. The figure shows $[5, 8]$ as 3, 3, 6, 6.
2. Node 1 $[1, 5]$; 2 $[1, 3]$; 3 $[4, 5]$; 4 $[1, 2]$; 5 $[3]$; 6 $[4]$; 7 $[5]$; 8 $[1]$; 9 $[2]$. The largest index is 9, below $4N = 20$. Here $2N = 10$ would also have sufficed, but for $N = 6$ the largest index is 13, above $2N = 12$, which is why $2N$ is not a safe size.
3. The range $[3, 8]$ splits into $[3, 4]$ (node 5) and $[5, 8]$ (node 3), both fully covered, so `query` returns them directly and combines them once.
4. Store the length of the range, the longest run anywhere (*best*), the run touching the left edge (*pre*), the run touching the right edge (*suf*), and the values at the two edges. When merging, a crossing run exists only if the right edge value of L equals the left edge value of R ; then it is $L.suf + R.pre$, and *pre* extends into R only when $L.pre$ equals the length of L .
5. The array 1, 1 with the query $[1, 2]$: each half has $ans = 1$, so the wrong program prints 1, while the true answer is 2.
6. A non-overlapping node would have to return an “empty” answer, but the node describes non-empty blocks, and there is no value of the four fields that behaves as a neutral element; combining with a made-up node would corrupt *pre*, *suf* and *ans*. The three-case query avoids ever asking such a node.

References

- [1] Sphere Online Judge, “GSS1 — Can you answer these queries I.” [Online]. Available: <https://www.spoj.com/problems/GSS1/> (accessed Sep. 2026).
- [2] J. Bentley, “Programming pearls: Algorithm design techniques,” *Communications of the ACM*, vol. 27, no. 9, pp. 865–873, 1984.
- [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] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.
- [5] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Lulu Press, 2020.

Chapter 3

Persistent Segment Trees and Order Statistics — MKTHNUM

How do you answer “what is the k -th smallest value among positions l to r ?” for thousands of ranges without sorting each range again? This chapter answers with one of the most elegant ideas in competitive programming: a segment tree that keeps *every* earlier version of itself, at a cost of only about $\log_2 N$ new nodes per version. You will learn coordinate compression, the frequency tree over values, the descent that finds the k -th smallest value in logarithmic time, persistence by *path copying*, and the trick that makes the whole chapter work: two versions of a tree can be subtracted, just as two prefix sums can. The case study, SPOJ MKTHNUM, puts all of it into fewer than sixty lines.

3.1 The Market Inspector’s Median

Every morning along the road to a hill town, durian sellers set up their stalls one after another, a long row numbered from the bottom of the hill to the top. A market inspector records each stall’s price for a kilogram of durian. Buyers, sellers and the local cooperative keep asking her questions of one shape: “among stalls 20 to 80, what is the middle price?” or “among the first thirty stalls, what is the third cheapest?” The middle price, the *median*, is a fairer summary than the average, because one stall with a wildly high price for a famous variety would pull the average up but barely moves the median.

For one question, she can copy the prices of the stalls in the range, sort them, and read position k . That costs about $n \log n$ steps for a range of n stalls. For a row of a hundred thousand stalls and five thousand questions, sorting each range costs up to $5\,000 \times 100\,000 \times 17 \approx 8.5 \times 10^9$ steps, much too slow. The SPOJ problem MKTHNUM [1] asks exactly this question at exactly this size.

3.2 The Idea in Plain Words

Suppose, for a moment, that the question were always about a *prefix*, stalls 1 to r . Then we could keep a *count of prices*: a table saying, for every possible price, how many of the first r stalls have it. To find the k -th smallest price, walk through the prices from low to high, adding counts, until the running total reaches k . A segment tree over the *prices* (not over the positions) makes that walk take $\log_2$ steps instead of one step per price.

Now add stalls one by one. The count table after stall r is the table after stall $r - 1$ plus one at the price of stall r . If we could keep all these tables, version 0 (empty) up to version N (everything), then a question about stalls l to r would be answered by *subtracting* version $l - 1$ from version r : what remains counts exactly the stalls l to r . Keeping $N + 1$ full trees would cost far too much memory, but consecutive versions differ only along one root-to-leaf path. *Path copying* stores just that path for each new version and shares everything else. That is the whole chapter.

3.3 Warm-Up Problems

3.3.1 Coordinate compression

Prices in MKTHNUM range from -10^9 to 10^9 , so a tree with one leaf per possible value is out of the question. But only N different values actually appear, and for the k -th smallest question only their *order* matters. *Coordinate compression* replaces each value by its rank among the distinct values: sort a copy, remove duplicates, and look each value up with binary search. **Listing 3.1** shows the three standard-library calls.

Listing 3.1 `compress.cpp`: sort, remove duplicates, and replace every value by its rank.

```
// Warm-up: coordinate compression with sort + unique + lower_bound.
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    vector<int> a(n);
    for (int& x : a)
        if (scanf("%d", &x) != 1) return 0;
    vector<int> vals = a;
    sort(vals.begin(), vals.end());
    vals.erase(unique(vals.begin(), vals.end()), vals.end());
    for (int x : a) { // rank of x among the distinct values, from 1
        int rank = (int)(lower_bound(vals.begin(), vals.end(), x) - vals.begin()) + 1;
        printf("%d ", rank);
    }
    printf("\n");
    for (int i = 0; i < (int)vals.size(); i++) printf("%d->%d ", i + 1, vals[i]);
    printf("\n");
    return 0;
}
```

On the values $10^9, -7, 42, -7, 5, 42$:

The first line gives the rank of each input value; the second line is the table to translate a rank back into a value. After compression, the values are the integers $1, \dots, U$, where $U \leq N$ is the number of distinct values, and a tree over $[1, U]$ has at most $4N$ nodes.

3.3.2 Subtracting two prefix counts

The second warm-up shows the “difference of versions” idea without any tree. When the values are tiny (1 to 9), we can afford to keep, for every prefix i , a whole count table $c_i[v]$, the number of positions among the first i whose value is v . Then the number of positions in $[l, r]$ with value v is

$$c_r[v] - c_{l-1}[v] \quad (3.1)$$

and the number of values at most x in $[l, r]$ is the sum of Equation (3.1) over $v \leq x$. **Listing 3.2** does this with a plain table.

Listing 3.2 `count_prefix.cpp`: counting values at most x in a range by subtracting two prefix count tables.

```
// Warm-up: "how many of a[l..r] are <= x?" by subtracting two prefix counts.
// Values are small (1..9), so cntUpTo[i][v] can be a plain table.
#include <cstdio>
using namespace std;

int cntUpTo[1001][10]; // cntUpTo[i][v] = how many of a[1..i] equal v

int main() {
    int n, m;
    if (scanf("%d %d", &n, &m) != 2) return 0;
    for (int i = 1; i <= n; i++) {
        int x;
        if (scanf("%d", &x) != 1) return 0;
        for (int v = 1; v <= 9; v++) cntUpTo[i][v] = cntUpTo[i - 1][v];
        cntUpTo[i][x]++; // version i = version i-1 plus one value
    }
    while (m--) {
        int l, r, x, total = 0;
        if (scanf("%d %d %d", &l, &r, &x) != 3) return 0;
        for (int v = 1; v <= x; v++) total += cntUpTo[r][v] - cntUpTo[l - 1][v];
        printf("%d\n", total);
    }
    return 0;
}
```

On the array 3, 1, 4, 1, 5, 9, 2 and the questions $(l, r, x) = (2, 5, 3), (1, 7, 4)$ and $(3, 6, 1)$:

In $[2,5]$ the values are 1, 4, 1, 5, and two of them are at most 3. Notice what the program does: version i of the table is version $i - 1$ plus one value, and a range question subtracts two versions. The persistent segment tree does exactly the same, with two differences: a tree instead of a flat table, so that it works for 10^5 distinct values, and shared nodes instead of copies, so that the $N + 1$ versions fit in memory.

3.4 A Segment Tree over Values

Build a segment tree over the value range $[1, U]$. Node v covering values $[lo, hi]$ stores $cnt(v)$, the number of inserted elements whose compressed value lies in $[lo, hi]$. The root's count is the number of elements inserted so far; a leaf's count is how many times its value was inserted. Inserting a value p adds one to every node on the path from the root to leaf p , and nothing else changes.

To find the k -th smallest inserted value, start at the root with k . At a node covering $[lo, hi]$ with middle mid , let $c = cnt(\text{left child})$ be the number of inserted values in $[lo, mid]$. If $k \leq c$, the k -th smallest is among them: go left with the same k . Otherwise the first c values are all smaller than the answer: go right looking for the $(k - c)$ -th smallest there:

$$kth(v, k) = \begin{cases} kth(\text{left}(v), k) & \text{if } k \leq cnt(\text{left}(v)) \\ kth(\text{right}(v), k - cnt(\text{left}(v))) & \text{otherwise} \end{cases} \quad (3.2)$$

and at a leaf the answer is the leaf's value. Equation (3.2) is binary search, but on counts rather than on a sorted array: each step halves the range of candidate values, so the descent takes $\lceil \log_2 U \rceil$ steps. Its correctness is an invariant: *the answer lies in the current node's value range, and it is the k -th smallest among the inserted values in that range*. It holds at the root, and each case of Equation (3.2) preserves it.

A small example makes the descent concrete. Insert the ranks 3, 1, 4, 1 into a tree over $[1,4]$. The root counts 4 values; its left child $[1,2]$ contains the two values 1, 1, so its count is 2, and the right child $[3,4]$ counts 3 and 4, also 2. To find the 3rd smallest, start at the root with $k = 3$. The left child holds only 2 values, fewer than 3, so the answer is in the right child, where we now look for the $(3 - 2) = 1$ st smallest. The left child of $[3,4]$ is the leaf $[3]$ with count 1, and $1 \leq 1$, so we go left and arrive at the leaf 3. Sorted, the values are 1, 1, 3, 4, and the 3rd is indeed 3. Every step discarded half of the remaining value range, and no step ever looked at an individual element.

3.5 Persistence by Path Copying

3.5.1 Copy the path, share the rest

An ordinary update overwrites nodes, and the old version is lost. A *persistent* update never overwrites. It creates a new copy of each node on the path from the root to the changed leaf, with the count increased by one, and lets each copy point to the new copy below it on the path and to the *old* child on the other side. The new root describes the new version; the old root still describes the old version, untouched. Driscoll, Sarnak, Sleator and Tarjan studied this and other ways of making data structures persistent in a classic paper [2].

Figure 3.1 shows the idea on four compressed values. Version 1 contains the value 2. Inserting the value 3 creates only three new nodes, the root $[1,4]$, the node $[3,4]$ and the leaf $[3]$, with counts 2, 1 and 1. The new root points left to the *old* node $[1,2]$, and the new $[3,4]$ points right to the old leaf $[4]$: both are shared, drawn dashed.

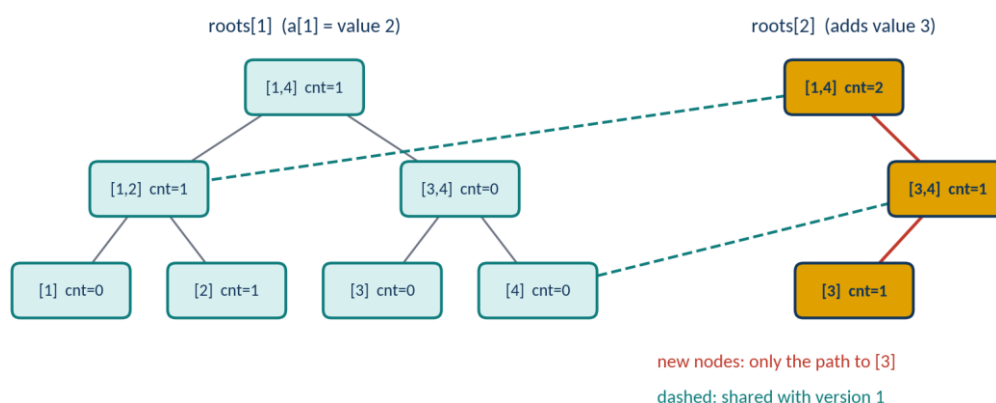


Figure 3.1 Path copying: inserting value 3 into version 1 creates only the three amber nodes on the path to leaf $[3]$; the dashed edges point to nodes shared with version 1.

3.5.2 How much memory

Each insertion creates one node per level, so $\lceil \log_2 U \rceil + 1$ nodes. With a single empty node for version 0, the total after N insertions is

$$\text{nodes} = 1 + N(\lceil \log_2 U \rceil + 1) \quad (3.3)$$

For $N = U = 100\,000$ that is $1 + 100\,000 \times 18 = 1\,800\,001$ nodes. With three `int` arrays (left child, right child, count) of 4 bytes each, the tree occupies about 21.6 MB, which is why the solution declares its arrays of size `MAXN * 18`. Equation (3.3) is the number to compute *before* writing the code: if it exceeds the memory limit, persistence is the wrong tool.

A useful detail: the empty tree does not need U nodes. A single node 0 with count 0 whose children are node 0 itself stands for “an empty subtree of any size”. A new version that goes left from an empty node copies node 0 and continues from node 0 again, so the empty parts of every version cost nothing.

3.5.3 Why sharing is safe

Sharing nodes between versions sounds dangerous: if two versions point to the same node, will a change made for one version not appear in the other? The answer is that *no node is ever changed after it has been created*. `insert` only reads the old nodes and writes into brand-new ones. A structure whose nodes never change after creation is called *immutable*, and immutable parts can be shared freely, exactly as two sentences can share a quotation without either one being able to alter it.

With that in mind, the correctness of version i follows by induction. Version 0 is the empty tree, and all its counts are zero, which is right for an empty prefix. Suppose version $i - 1$ is a correct count tree for $a_1, \dots, a_{i-1}$. The insertion of the rank p of a_i creates new copies of exactly the nodes whose value range contains p , each with its count increased by one, which is exactly the change that a_i makes to the counts. Every other node position keeps the node of version $i - 1$, whose count is still right because a_i does not fall in its range. So version i is a correct count tree for $a_1, \dots, a_i$, and the induction is complete.

The same argument explains why the versions may be built only in order and never changed afterwards. MKTHNUM has no updates, so this is no restriction here. When a problem mixes range questions with updates in the middle of the array, a plain persistent tree is no longer enough, and heavier tools (a Fenwick tree whose cells are trees, or offline processing) take over.

3.6 Subtracting Two Versions

Build version i by inserting a_i into version $i - 1$, for $i = 1, \dots, N$, and keep the roots $root_0, \dots, root_N$. Because every insertion only *adds* one to counts, the count of any node position in version r minus its count in version $l - 1$ is the number of elements among $a_l, \dots, a_r$ whose value falls in that node’s range:

$$cnt_{[l,r]}(v) = cnt(v \text{ in version } r) - cnt(v \text{ in version } l - 1) \quad (3.4)$$

Equation (3.4) is Equation (3.1) lifted into the tree. The two versions have the same *shape* (the same value ranges at the same positions), even though they share different nodes, so we can walk down both at once: at each step, compute the left count of the range with Equation (3.4), and apply Equation (3.2) to it, moving both pointers to their left children or both to their right children.

Let us trace the query “3rd smallest in positions 2 to 5” on the array 1, 5, 2, 6, 3, 7, 4, whose values are already the ranks 1 to 7. The range holds 5, 2, 6, 3.

- Root, values [1,7], middle 4. The left child counts values 1 to 4: 3 of them in version 5 (1, 2, 3), 1 in version 1 (1). Difference 2 (the values 2 and 3). Since $k = 3 > 2$, go right with $k = 1$.
- Node [5,7], middle 6. Left child [5,6]: version 5 has 2 (5, 6), version 1 has 0. Since $k = 1 \leq 2$, go left.
- Node [5,6], middle 5. Left child [5]: difference 1. Since $k = 1 \leq 1$, go left.
- Leaf [5]: the answer is the value 5.

Sorted, the range is 2, 3, 5, 6, and its third element is indeed 5.

KEY INSIGHT: Persistence turns a problem about *ranges of positions* into a problem about *two prefixes*, and additivity lets you subtract them. The same pattern answers “how many values in $[l, r]$ are at most x ”, “the rank of x in $[l, r]$ ”, and many more.

3.7 HARD Case Study: MKTHNUM

3.7.1 The problem in our own words

You are given an array $a_1, \dots, a_N$ of distinct integers of absolute value up to 10^9 , with $N \leq 100\,000$, and $M \leq 5\,000$ questions (i, j, k) . For each question, imagine the part $a_i, \dots, a_j$ sorted in increasing order, and print its k -th element, where $1 \leq k \leq j - i + 1$. The input gives N and M , then the array, then one question per line.

The statement promises distinct values, but our solution does not rely on it: duplicates simply give a leaf a count larger than one, and Equation (3.2) still works. Robustness like this costs nothing and protects you when a statement is less precise than you think.

3.7.2 Sample input and output

```
7 3
1 5 2 6 3 7 4
2 5 3
4 4 1
1 7 3
```

The first question is the one traced in Section 3.6. The second asks for the only element of $[4,4]$, which is 6. The third asks for the 3rd smallest of the whole array: 1, 2, 3, so 3.

3.7.3 Design

The plan follows the chapter. Compress the values (Section 3.3.1). Create version 0 as a single empty node. For $i = 1$ to N , insert the rank of a_i into version $i - 1$ with path copying to obtain version i . For each question, walk down versions $i - 1$ and j together with Equation (3.2) and Equation (3.4), and translate the leaf’s rank back into a value. Building costs $O(N \log N)$ time and the memory of Equation (3.3); each question costs $O(\log N)$, so the total is $O((N + M) \log N)$, a few million steps.

3.7.4 The full solution

Listing 3.3 is the complete program. It has three short functions, and only one of them, `kth`, knows anything about the question.

Listing 3.3 `MKTHNUM.cpp`: persistent segment tree over compressed values; each question walks down two versions.

```
// MKTHNUM (SPOJ): k-th smallest value of a[i..j], many queries.
// Persistent segment tree over compressed values: version r holds a[1..r].
#include <algorithm>
#include <cstdio>
using namespace std;

const int MAXN = 100001;
const int MAXNODES = MAXN * 18 + 5; // one empty root + about Log2(n)+1 per insert
int lc[MAXNODES], rc[MAXNODES], cnt[MAXNODES];
int nodeCount = 0; // node 0 is the shared "empty" node: lc = rc = cnt = 0
int roots[MAXN], a[MAXN], vals[MAXN];

int newNode() {
    ++nodeCount;
    lc[nodeCount] = rc[nodeCount] = cnt[nodeCount] = 0;
    return nodeCount;
}

// Returns the root of a new version: old version prev plus one value at pos.
// Only the nodes on the path to pos are copied; all others are shared.
int insert(int prev, int lo, int hi, int pos) {
    int cur = newNode();
    lc[cur] = lc[prev];
    rc[cur] = rc[prev];
    cnt[cur] = cnt[prev] + 1;
    if (lo == hi) return cur;
    int mid = (lo + hi) / 2;
    if (pos <= mid) lc[cur] = insert(lc[prev], lo, mid, pos);
    else rc[cur] = insert(rc[prev], mid + 1, hi, pos);
    return cur;
}

// <P50>
// kth: walk down version vr and version vl together; the counts of their left
// children differ by how many of a[l..r] are in the lower half of the values.
// Go left if k fits there, otherwise go right with k reduced; return the leaf.
// >>> YOUR TURN (Public Edition): write this part yourself, following the hint above. <<<
// </P50>
```

Listing 3.3 (Continued) MKTHNUM.cpp: persistent segment tree over compressed values; each question walks down two versions.

```
int main() {
    int n, m;
    if (scanf("%d %d", &n, &m) != 2) return 0;
    for (int i = 1; i <= n; i++) {
        if (scanf("%d", &a[i]) != 1) return 0;
        vals[i] = a[i];
    }
    sort(vals + 1, vals + n + 1); // coordinate compression: 1..un
    int un = (int)(unique(vals + 1, vals + n + 1) - (vals + 1));
    roots[0] = newNode(); // version 0: no values yet
    for (int i = 1; i <= n; i++) {
        int pos = (int)(lower_bound(vals + 1, vals + un + 1, a[i]) - vals);
        roots[i] = insert(roots[i - 1], 1, un, pos);
    }
    while (m--) {
        int l, r, k;
        if (scanf("%d %d %d", &l, &r, &k) != 3) return 0;
        printf("%d\n", vals[kth(roots[l - 1], roots[r], 1, un, k)]);
    }
    return 0;
}
```

3.7.5 Line by line

Storage. Instead of a `struct` with pointers, the tree lives in three parallel arrays, `lc`, `rc` and `cnt`, indexed by node number. This is compact (12 bytes per node) and fast. `MAXNODES` follows Equation (3.3), with five spare entries. Node 0 is never created by `newNode`; as a global it is all zeros, so it is the shared empty node whose children are itself.

newNode. It hands out the next free index and clears its fields. Because the arrays are global and the program handles one test, the clearing is not strictly necessary, but it keeps the function correct if you reuse it for several test cases.

insert. `insert(prev, lo, hi, pos)` returns the root of a new version equal to `prev` plus one value at `pos`. It copies `prev` into a fresh node `cur` with the count increased, and if `cur` is not a leaf, it replaces exactly one child: the one on the side of `pos`, by a recursive call on the corresponding child of `prev`. The other child pointer still points into the old version: that is the sharing of Figure 3.1.

kth. Between the markers `// <P50>` and `// </P50>`, `kth(vl, vr, lo, hi, k)` walks down two versions at once, `vl` for version $l - 1$ and `vr` for version r . `leftCount` is Equation (3.4) applied to the left children, and the two branches are Equation (3.2). The recursion stops at a leaf and returns its rank.

main. It reads the array, keeping a copy in `vals`, and compresses: `sort` then `unique` leave the U distinct values in `vals[1..un]`, and `lower_bound` returns the position of each a_i among them, which is its rank. Version 0 is one new empty node; then each version is built from the previous one. Each question prints `vals[kth(roots[l - 1], roots[r], 1, un, k)]`, turning the rank back into the original value.

3.7.6 Traps

- **Forgetting compression.** A tree over $[-10^9, 10^9]$ would need a depth of 31 and far more nodes; compressing first is mandatory.
- **Using version l instead of $l - 1$.** Subtracting version l removes a_l from the range. The sample catches this only by luck; the stress test catches it at once.
- **An undersized node pool.** Compute Equation (3.3). With a pool of `MAXN * 17`, the program works on small inputs and writes past the end of the arrays on the largest ones.
- **Printing the rank.** The walk returns a *rank*; the answer is `vals[rank]`.
- **Walking two versions of different shapes.** Both versions must be built over the same value range $[1, U]$. Building each version over “the values seen so far” would break Equation (3.4).

3.7.7 Stress test

Listing 3.4 answers each question the obvious way: copy the range, sort it, read position k .

Listing 3.4 `brute.cpp`: sort a copy of each range, $O(n \log n)$ per question.

```
// Brute force for MKTHNUM: copy a[l..r], sort it, read position k.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n, m;
    cin >> n >> m;
    vector<int> a(n + 1);
    for (int i = 1; i <= n; i++) cin >> a[i];
    while (m--) {
        int l, r, k;
        cin >> l >> r >> k;
        vector<int> part(a.begin() + l, a.begin() + r + 1);
        sort(part.begin(), part.end());
        cout << part[k - 1] << "\n";
    }
    return 0;
}
```

The generator **Listing 3.5** produces arrays at three value scales; the tiny scale (± 3) creates many duplicates, which tests the robustness promised in Section 3.7.1.

Listing 3.5 `gen.py`: random MKTHNUM inputs, with and without duplicates.

```
#!/usr/bin/env python3
"""Random MKTHNUM input: python3 gen.py SEED"""
import random
import sys

rng = random.Random(int(sys.argv[1]) if len(sys.argv) > 1 else 1)
n, m = rng.randint(1, 12), rng.randint(1, 10)
lim = rng.choice([3, 20, 10**9])
print(n, m)
print(*[rng.randint(-lim, lim) for _ in range(n)])
for _ in range(m):
    i = rng.randint(1, n)
    j = rng.randint(i, n)
    print(i, j, rng.randint(1, j - i + 1))
```

```
all 500 tests identical
```

On the largest input, $N = 100\,000$ random values of absolute value up to 10^9 and $M = 5\,000$ random questions, the solution runs in about 0.09 seconds on our machine; most of that time is spent reading the input and sorting for compression.

3.7.8 The cost in numbers

It is worth checking the analysis against the real limits, because this is the habit that prevents time-limit surprises. Reading 10^5 numbers with `scanf` takes a few milliseconds. Sorting them for compression costs about $N \log_2 N \approx 1.7 \times 10^6$ comparisons. Building the N versions creates 1.8×10^6 nodes, each with a handful of array writes. Each of the 5 000 questions walks 18 levels, touching four array cells per level: under 4×10^5 memory accesses in total. Every one of these numbers is far below the 10^8 operations per second of Table 1.2 in Chapter 1, which is consistent with the measured 0.09 seconds.

Memory is the tighter resource. The three arrays of `MAXNODES` integers occupy about 21.6 MB, and the other arrays another 1.2 MB. That is comfortable on SPOJ and in the national round, but it is the kind of number you should write in the margin of your paper before coding, since doubling it

(for example by storing 64-bit sums in the nodes) or multiplying it by the number of test cases can suddenly matter.

3.8 Relatives and Alternatives

The same persistent tree answers other questions with the same walk: the number of values at most x in $[l, r]$ (walk towards the rank of x and add left counts), the rank of x , or the sum of the k smallest values (store a sum next to each count). Other tools solve the k -th smallest problem too, and a contestant should know at least their names and trade-offs:

- A *merge sort tree* stores, in each node of an ordinary segment tree over positions, the sorted list of its elements. A count query takes $O(\log^2 N)$, and a k -th query needs an extra binary search on the value, $O(\log^3 N)$. It is simpler to reason about but slower.
- A *wavelet tree* answers the k -th smallest in $O(\log U)$ with less memory than persistence; Navarro's survey [3] is the standard introduction.
- Offline methods, such as sorting the questions and processing them with a Fenwick tree and parallel binary search, avoid persistence altogether when all questions are known in advance.

For the classical order-statistic problem on a single array (the k -th smallest of everything), linear-time selection algorithms exist; Cormen et al. [4] present both the randomized and the worst-case linear methods. The persistent tree is what makes the *range* version fast. Textbooks aimed at contestants, such as Laaksonen [5] and Halim et al. [6], treat persistence as an advanced but standard technique, and it appears in national and IOI selection problems regularly.

3.9 Contest-Day Tactics

- **Recognize the pattern.** “Range”, “ k -th smallest”, “median”, “how many values below x in a range”, with values too large for a table: think of compression plus a persistent tree.
- **Compute memory first.** Apply Equation (3.3) before writing anything. If the product of N and the depth is too big, look at the offline alternatives of Section 3.8.
- **Bank the sorting subtask.** Listing 3.4 solves small subtasks and becomes the stress-test reference.
- **Use arrays, not pointers.** Parallel arrays with integer indices are faster to write, easier to debug, and avoid `new` in a loop.
- **Test with duplicates** even when the statement promises distinct values.

3.10 Summary

- Coordinate compression maps up to N distinct values onto ranks $1, \dots, U$ with `sort`, `unique` and `lower_bound`.
- A segment tree over values with counts finds the k -th smallest inserted value by the descent Equation (3.2) in $O(\log U)$ steps.
- Path copying keeps every version of the tree by copying only the $\lceil \log_2 U \rceil + 1$ nodes on the updated path; the memory is given by Equation (3.3).
- Because insertions are additive, subtracting version $l - 1$ from version r (Equation (3.4)) gives the counts of the range $[l, r]$, and the descent walks both versions together.
- MKTHNUM is solved in $O((N + M)\log N)$ time with about 1.8×10^6 nodes.

3.11 Exercises

1. Compress the values 40, -3, 40, 17, 0 by hand. Which ranks do you get, and what is U ?
2. How many nodes does Equation (3.3) give for $N = 50\,000$ distinct values? How many bytes with three `int` arrays?
3. Trace the walk for the question “1st smallest in positions 1 to 7” on the sample array.
4. Modify `kth` into a function `countAtMost(vl, vr, lo, hi, x)` that returns how many values of the range have rank at most x . Write its recursion in words.
5. Why must version 0 exist, instead of starting with version 1?
6. The array is 2, 2, 2 and the question is (1,3,2). Does the solution work, and what does it print?
7. Extend the node so that a question can also ask for the *sum* of the k smallest values of $a_i, \dots, a_j$. What must each node store, and how does the walk change?

3.12 Solutions to the Exercises

1. The distinct sorted values are -3, 0, 17, 40, so $U = 4$ and the ranks are 4, 1, 4, 3, 2.
2. $\lceil \log_2 50\,000 \rceil = 16$, so $1 + 50\,000 \times 17 = 850\,001$ nodes, and $850\,001 \times 12 \approx 10.2$ MB.
3. Versions 0 and 7. At [1,7] the left child [1,4] has count $4 - 0 = 4 \geq 1$: go left. At [1,4] the left child [1,2] has count 2: go left. At [1,2] the left child [1] has count 1: go left. The leaf is rank 1, so the answer is the value 1.
4. At a leaf, return the leaf’s count difference if its rank is at most x , else 0. At an inner node with middle mid : if $x \leq mid$, return the result for the left children only; otherwise return the whole left count difference (every value there is at most x) plus the result for the right children. It visits $O(\log U)$ nodes.
5. A question with $l = 1$ subtracts version $l - 1 = 0$. Version 0 is the empty tree, whose counts are all zero; without it, questions starting at position 1 would need a special case.
6. It works: there is one distinct value, $U = 1$, and every version is a single leaf with count 1, 2 or 3. `kth` returns rank 1 immediately, and the program prints 2.
7. Each node also stores *sum*, the total of the original values inserted into its value range, as a `long long` (up to 10^5 values of size 10^9 exceed 32 bits). `insert` adds the inserted value to every copied node. During the walk, whenever the descent turns right, the whole left child of the range is among the k smallest, so its sum difference (version r minus version $l - 1$) is added to an accumulator. At the leaf, the remaining k copies of the leaf’s value are added. The walk stays $O(\log U)$, and the memory grows by 8 bytes per node, to about 36 MB for $N = 10^5$.

References

- [1] Sphere Online Judge, “MKTHNUM — K-th Number.” [Online]. Available: <https://www.spoj.com/problems/MKTHNUM/> (accessed Sep. 2026).
- [2] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, “Making data structures persistent,” *Journal of Computer and System Sciences*, vol. 38, no. 1, pp. 86–124, 1989.
- [3] G. Navarro, “Wavelet trees for all,” *Journal of Discrete Algorithms*, vol. 25, pp. 2–20, 2014.
- [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] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

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

Chapter 4

Monotonic Stacks and Linear Array Techniques — HISTOGRAM

Some problems with N up to a hundred thousand look as if they need heavy machinery, and the answer turns out to be a single left-to-right pass with a stack kept in sorted order. This chapter builds that tool, the *monotonic stack*, from the ground up. You will learn to compute, for every element of an array, its nearest smaller neighbour on each side in one pass; to prove, with a simple counting argument, that a loop nested inside a loop can still run in linear time; and to recognize the family of problems where each element's answer is decided by the first neighbour that breaks a condition. The case study is the world classic HISTOGRAM, the largest rectangle in a histogram, a favourite of olympiad juries because it is short to state, easy to solve slowly, and beautiful to solve in $O(N)$.

4.1 The Banner on Main Street

For the Independence Day celebration on the 17th of August, a village wants to hang the largest possible rectangular banner along the front of the buildings on its main street. The buildings stand side by side, each one exactly one unit wide, and they have different heights: a two-storey shop, a one-storey warung, a tall mosque tower, a small house. The banner must be a rectangle resting on the ground line, and it may not rise above any building it covers. Which rectangle has the largest area?

Try it on seven buildings with heights 2, 1, 4, 5, 1, 3, 3. A banner over the single tallest building has area $5 \times 1 = 5$. A banner over the two tall buildings in the middle has height 4 (it may not rise above the shorter one) and width 2: area 8. A banner over the whole street has height 1 and width 7: area 7. The best is 8, but finding it for a street of a hundred thousand buildings is another matter. There are about $N^2/2$ ways to choose the first and last building of the banner, five billion for $N = 10^5$. That is the SPOJ problem HISTOGRAM [1].

4.2 The Idea in Plain Words

Look at the best banner. Its top edge touches some building, the lowest building it covers; otherwise the banner could be raised a little and be larger. So the best banner is, for some building i , the widest banner of height exactly h_i that contains building i . How wide can it be? It extends to the left until the first building shorter than h_i , and to the right until the first building shorter than h_i . If L_i is the index of the nearest strictly shorter building on the left (or -1 if there is none) and R_i the nearest strictly shorter on the right (or N if none), with buildings numbered 0 to $N - 1$, then the banner for building i covers the buildings strictly between them, and

$$area_i = h_i \cdot (R_i - L_i - 1) \quad (4.1)$$

The answer is the largest $area_i$. So the whole problem reduces to computing the *nearest smaller element* on each side of every position, and that is what a monotonic stack does in linear time.

4.2.1 Two slower roads to the same answer

Before the fast method, it helps to see the slower ones, because in a contest they are the subtask solutions and the stress-test references. The most direct method fixes the first bar i of the rectangle and moves the last bar j to the right, keeping the running minimum height m of bars i to j ; the best rectangle spanning exactly these bars has area $m \cdot (j - i + 1)$. That is $O(N^2)$ time: fine for $N \leq 5\,000$, hopeless for 10^5 .

A cleverer road is *divide and conquer*. The lowest bar of the whole histogram, at position p , splits the problem: the best rectangle either spans the whole histogram at height h_p (area $h_p \cdot N$), or lies entirely to the left of p , or entirely to the right of p , because no rectangle can cross bar p higher than h_p . Solve the two sides recursively and take the best of three. With a sparse table or segment tree to find the minimum of a range, each call costs $O(\log N)$ or $O(1)$, and there are N calls, so the method runs in $O(N \log N)$ when the splits are balanced. On a sorted histogram, however, the recursion depth becomes N , which can overflow the call stack for $N = 10^5$. The monotonic stack avoids both the logarithm and the deep recursion.

4.3 Warm-Up Problems

4.3.1 Nearest smaller elements

Walk from left to right and keep a stack of indices whose heights increase from the bottom to the top. When building i arrives, every index on top of the stack whose height is at least h_i can never again be anyone's nearest smaller element on the left: building i is at least as low and is closer to everything that comes later. So pop them. Whatever is left on top is the nearest building strictly lower than h_i , which is L_i . Then push i . The right side is the mirror image, a pass from right to left.

Listing 4.1 computes both, and prints Equation (4.1) for every building.

Listing 4.1 `nearest_smaller.cpp`: nearest strictly smaller element on each side, two passes with a stack.

```
// Warm-up: for every bar, the nearest strictly smaller bar on each side.
#include <stdio>
#include <vector>
using namespace std;

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    vector<long long> h(n);
    for (auto& x : h)
        if (scanf("%lld", &x) != 1) return 0;
    vector<int> L(n), R(n), st;
    for (int i = 0; i < n; i++) { // Left to right
        while (!st.empty() && h[st.back()] >= h[i]) st.pop_back();
        L[i] = st.empty() ? -1 : st.back(); // -1: nothing smaller on the left
        st.push_back(i);
    }
    st.clear();
    for (int i = n - 1; i >= 0; i--) { // right to left, the mirror image
        while (!st.empty() && h[st.back()] >= h[i]) st.pop_back();
        R[i] = st.empty() ? n : st.back(); // n: nothing smaller on the right
        st.push_back(i);
    }
    for (int i = 0; i < n; i++)
        printf("i=%d h=%lld L=%d R=%d area=%lld\n", i, h[i], L[i], R[i],
            h[i] * (R[i] - L[i] - 1));
    return 0;
}
```

On the street of Section 4.1:

Building 2 (height 4) has its nearest shorter neighbours at positions 1 and 4, so its banner covers positions 2 and 3: area 8, the answer. Building 1 and building 4 (both of height 1) have no strictly shorter building anywhere, so each can stretch over the whole street: area 7.

4.3.2 Stock span

A second classic uses a stack that *decreases*. The *span* of a day's share price is the number of consecutive days, ending today, on which the price was at most today's. A stack of pairs (price, span) with strictly decreasing prices answers it: when today's price arrives, every stacked day with a price at most today's is swallowed, and its span is added to today's. **Listing 4.2** shows the whole program.

Listing 4.2 `stock_span.cpp`: stock spans with a decreasing stack of (price, span) pairs.

```
// Warm-up: stock span, how many consecutive days up to today had a price
// no higher than today's. A decreasing stack of (price, span) pairs.
#include <cstdio>
#include <utility>
#include <vector>
using namespace std;

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    vector<pair<int, int>> st; // prices strictly decrease from bottom to top
    for (int i = 0; i < n; i++) {
        int price, span = 1;
        if (scanf("%d", &price) != 1) return 0;
        while (!st.empty() && st.back().first <= price) {
            span += st.back().second; // swallow the days that bar already covered
            st.pop_back();
        }
        st.push_back({price, span});
        printf("%d ", span);
    }
    printf("\n");
    return 0;
}
```

For the prices 100, 80, 60, 70, 60, 75, 85:

On the sixth day the price 75 swallows the pairs (60, 1) and (70, 2), which themselves had already swallowed an earlier 60; the span is $1 + 1 + 2 = 4$. The stack remembers, in compressed form, everything that the future can still need.

4.4 The Monotonic Stack

4.4.1 The invariant

In both warm-ups, the stack satisfies an *invariant*, a statement that is true before and after every step of the loop. For nearest smaller elements it is: *the stack holds indices in increasing order of position, with strictly increasing heights, and they are exactly the positions seen so far that have no strictly lower position to their right among the positions seen so far*. Those are the only positions that can still be the nearest smaller element of a future building. When a new building i arrives, the positions that lose this property are exactly those with heights at least h_i , and they are on top of the stack, because the heights increase towards the top. Popping them and pushing i restores the invariant.

Seen geometrically, the stack is the “visible staircase” when you look to the left from the current building: the lowest building, then the next lowest to its right that is taller, and so on up to the current one.

4.4.2 Why a loop inside a loop is linear

The code has a `while` loop inside a `for` loop, which looks quadratic. It is not, and the argument is worth learning because it applies to many algorithms. Count the operations on the stack instead of the iterations of the loops. Every index is pushed exactly once. Every index is popped at most once, because once popped it never returns. Each iteration of the inner `while` loop pops one index. Hence, over the whole run,

$$\text{pushes} + \text{pops} \leq N + N = 2N \quad (4.2)$$

and the total work is $O(N)$, even though a single step may pop many indices. This way of charging the cost of an expensive step to earlier cheap steps is called *amortized analysis*; Cormen et al. [2] present it in general, and Tarjan’s paper [3] is the classic source. The important habit is to ask “how many times can each element be touched in total?” rather than “how long can one step take?”

4.4.3 Four variants, one pattern

The same loop answers four related questions, depending on the order kept in the stack and on the comparison that triggers a pop. **Table 4.1** lists them for a left-to-right pass; a right-to-left pass gives the neighbours on the right. It is worth memorizing the table as a whole rather than four separate programs: the stack keeps the elements that are *better candidates* than everything above them, and the pop condition removes exactly the elements that the newcomer makes useless.

Table 4.1 The four nearest-neighbour questions and the pop condition that answers each in one left-to-right pass.

Question for position i (looking left)	Stack order, bottom to top	Pop while the top is
nearest strictly smaller	strictly increasing	$\geq h_i$
nearest smaller or equal	non-decreasing	$> h_i$
nearest strictly greater	strictly decreasing	$\leq h_i$
nearest greater or equal	non-increasing	$< h_i$

The first row is **Listing 4.1**; the stock span of **Listing 4.2** is the third row, since the day that stops a span is the nearest day with a strictly greater price. When a problem needs “the nearest smaller” on one side and “the nearest smaller or equal” on the other, as the sum-of-subarray-minimums problem does to avoid counting a subarray twice when values repeat, the table tells you immediately which pop condition to write in each pass.

4.5 One Pass: Answers at Pop Time

The two-pass method of **Listing 4.1** already solves the problem in linear time. There is a neater single pass, and it is the one used in the case study. Watch the moment when building j is popped because a building i with $h_i < h_j$ arrived:

- building i is the first building to the right of j that is strictly lower, so $R_j = i$;
- the index just below j in the stack is the nearest building to the left of j that is lower than or equal to h_j , so it plays the role of L_j (if the stack becomes empty, there is none, and the left end is -1).

At that moment, then, everything needed for **Equation (4.1)** is known, and the candidate area of j is computed right there:

$$\text{width}_j = \begin{cases} i & \text{if the stack is empty after popping } j \\ i - t - 1 & \text{where } t \text{ is the new top} \end{cases} \quad (4.3)$$

To make sure every building is eventually popped, a *sentinel* of height 0 is processed after the last building, at position N . It is lower than everything, so it empties the stack, and no special case is needed after the loop.

Figure 4.1 shows the key moment on our street: building 4, of height 1, pops building 3 (height 5, width 1, area 5) and then building 2 (height 4). After popping building 2, the new top is building 1, so the width is $4 - 1 - 1 = 2$ and the area is 8.

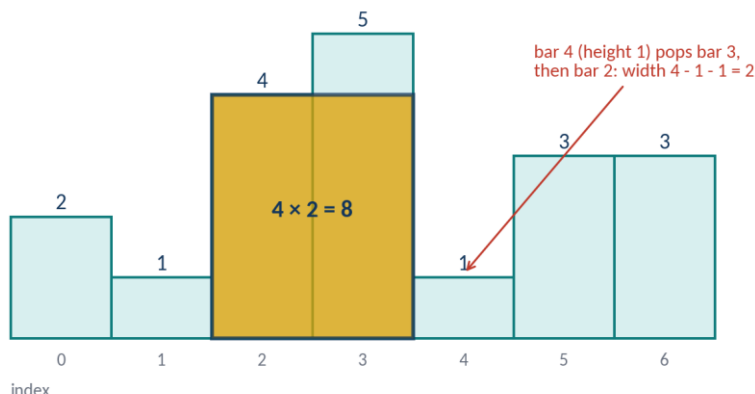


Figure 4.1 The histogram 2, 1, 4, 5, 1, 3, 3 and its largest rectangle; the rectangle is found when bar 4 pops bar 2 from the stack.

Table 4.2 traces the whole pass. Each row is one value of i ; the stack is written bottom to top as indices.

Table 4.2 The single-pass algorithm on 2, 1, 4, 5, 1, 3, 3; every candidate area is computed at the moment its bar is popped.

i	Height	Popped: (index, height, width, area)	Stack after	Best
0	2	none	0	0
1	1	(0, 2, 1, 2)	1	2
2	4	none	1, 2	2
3	5	none	1, 2, 3	2
4	1	(3, 5, 1, 5), (2, 4, 2, 8)	1, 4	8
5	3	none	1, 4, 5	8
6	3	none (equal height stays)	1, 4, 5, 6	8
7	0 (sentinel)	(6, 3, 1, 3), (5, 3, 2, 6), (4, 1, 5, 5), (1, 1, 7, 7)	7	8

Look at the last row. Building 6 is popped first with width 1, although a height-3 banner could cover buildings 5 and 6. That is harmless: building 5 is popped next with width 2 and area 6, which covers the same banner. With equal heights, the *earlier* bar gets the full width. Likewise building 4 gets width 5 while building 1, with the same height, gets the full width 7. The single pass does not compute every $area_i$ of Equation (4.1) exactly, but it always computes the correct value for at least one bar of each maximal banner, which is all we need for the maximum.

4.5.1 Why the single pass finds the maximum

Let the best rectangle have height H and cover bars a to b . Its height equals the lowest bar in $[a, b]$, and since it cannot be widened, bar $a - 1$ (if it exists) and bar $b + 1$ (if it exists) are strictly lower than H . Among the bars in $[a, b]$ with height exactly H , take the *leftmost*, say j . Every bar between a and $j - 1$ is strictly higher than H .

Now follow bar j through the pass. It is pushed at time j . It cannot be popped before time $b + 1$, because every bar in $(j, b]$ has height at least H , and the pass pops only for strictly lower bars. It is popped at time $b + 1$ (bar $b + 1$ is lower, or $b + 1 = N$ is the sentinel). At that moment, what lies

below j in the stack? Every bar between a and $j - 1$ is higher than h_j , so each of them was popped by j at time j , or earlier by something else. Bar $a - 1$, being lower than H , is still there, because the bars in $[a, b]$ all have height at least H and cannot pop it. So the new top after popping j is exactly $a - 1$. The width computed by Equation (4.3) is therefore $(b + 1) - (a - 1) - 1 = b - a + 1$, and the area computed for j is exactly $H \cdot (b - a + 1)$, the optimum. Every other computed area is the area of some real rectangle, so it can never exceed the optimum. Hence the maximum of the computed areas is the answer.

This style of proof, following one well-chosen element through the algorithm, is the standard way to argue about stack algorithms, and it is worth writing out once by hand.

4.6 HARD Case Study: HISTOGRAM

4.6.1 The problem in our own words

The input contains several test cases, one per line. Each line starts with n ($1 \leq n \leq 100\,000$) followed by n heights $h_1, \dots, h_n$ with $0 \leq h_i \leq 10^9$; a line containing only 0 ends the input. The histogram consists of n bars of width 1 standing side by side on a common baseline. For each test case, print the area of the largest axis-parallel rectangle that lies entirely inside the histogram.

4.6.2 Sample input and output

```
7 2 1 4 5 1 3 3
4 1000 1000 1000 1000
0
```

The first line is our street. The second is four bars of height 1000: the whole histogram is a rectangle of area 4000.

4.6.3 Design

The bounds (Table 1.2 in Chapter 1) allow $O(n \log n)$ or $O(n)$ per test. The single pass of Section 4.5 is $O(n)$ by Equation (4.2). Two sizes need care. A height reaches 10^9 and a width 10^5 , so an area reaches 10^{14} , far beyond 32 bits: heights, widths and areas are `long long`. And since the number of test cases is not given, the program must loop until it reads the 0.

4.6.4 The full solution

Listing 4.3 is the complete program; its core is eight lines.

Listing 4.3 `HISTOGRAM.cpp`: one pass with a monotonic stack and a height-0 sentinel.

```
// HISTOGRAM (SPOJ): Largest rectangle in a histogram, O(n) per test.
// Monotonic stack of indices whose heights increase from bottom to top.
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

long long h[100001];

int main() {
    int n;
    while (scanf("%d", &n) == 1 && n != 0) {
        for (int i = 0; i < n; i++)
            if (scanf("%lld", &h[i]) != 1) return 0;
        vector<int> st; // indices; h[st[0]] <= h[st[1]] <= ... (bottom to top)
        st.reserve(n);
        long long best = 0;
        for (int i = 0; i <= n; i++) {
            long long cur = (i < n) ? h[i] : 0; // bar n is a height-0 sentinel
            // <P50>
            // Pop every bar taller than cur: cur is its first shorter bar on the
            // right, the new top is its first shorter-or-equal bar on the left, so
            // its widest rectangle spans the gap in between. Then push i.
            // >>> YOUR TURN (Public Edition): write this part yourself, following the hint
            // above. <<<
            // </P50>
        }
        printf("%lld\n", best);
    }
    return 0;
}
```

4.6.5 Line by line

Reading. `while (scanf("%d", &n) == 1 && n != 0)` handles both the terminating 0 and an unexpected end of input. Heights are read into the global `long long` array `h`, so products never overflow.

The stack. A `vector<int>` used as a stack (`push_back`, `back`, `pop_back`) is as fast as `std::stack` and lets us `reserve` memory once. It holds indices, never heights; the heights are looked up in `h`.

The loop. `i` runs from 0 to n inclusive. For $i = n$, `cur` is the sentinel height 0.

Popping. Between the markers `// <P50>` and `// </P50>`, the `while` loop pops every bar strictly taller than `cur`. For each popped bar it reads the bar's height, pops it, computes the width by Equation (4.3), and updates the best area. Then `i` is pushed. The comparison is strict (`>`), so bars of equal height stay on the stack; Section 4.5 showed why that is correct.

Output. After the pass, `best` is printed with `%lld`.

4.6.6 Traps

- **32-bit areas.** $10^9 \times 10^5$ does not fit in an `int`; even with `long long` heights, writing `int width` and multiplying two `ints` overflows before the conversion.
- **Forgetting the sentinel.** Without the closing height-0 bar, the bars still on the stack at the end are never processed; on an increasing histogram such as 1, 2, 3 the program would print 0.
- **The wrong width when the stack is empty.** Then the popped bar is the lowest so far and extends to the left edge: the width is `i`, not `i - 1`.

- **Reading the whole input as one test.** Each line is a separate test; the loop must restart the stack and the best area for each.
- **Zero heights.** A bar of height 0 is legal. It pops everything, and its own candidate area is 0; nothing special is needed, but include it in your tests.

4.6.7 Stress test

Listing 4.4 checks every pair (i, j) with a running minimum: $O(n^2)$ and obviously correct.

Listing 4.4 `brute.cpp`: every pair of first and last bar, with a running minimum height.

```
// Brute force for HISTOGRAM: every pair (i, j) with the running minimum.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n;
    while (cin >> n && n != 0) {
        vector<long long> h(n);
        for (auto& x : h) cin >> x;
        long long best = 0;
        for (int i = 0; i < n; i++) {
            long long low = h[i];
            for (int j = i; j < n; j++) {
                low = min(low, h[j]);
                best = max(best, low * (j - i + 1));
            }
        }
        cout << best << "\n";
    }
    return 0;
}
```

The generator **Listing 4.5** writes one to four test lines followed by 0; the heights come from four scales, including all-zero histograms and the full 10^9 .

Listing 4.5 `gen.py`: random HISTOGRAM inputs, several test lines ending with 0.

```
#!/usr/bin/env python3
"""Random HISTOGRAM input (several tests, then 0): python3 gen.py SEED"""
import random
import sys

rng = random.Random(int(sys.argv[1]) if len(sys.argv) > 1 else 1)
for _ in range(rng.randint(1, 4)):
    n = rng.randint(1, 12)
    lim = rng.choice([0, 3, 10, 10**9])
    print(n, *[rng.randint(0, lim) for _ in range(n)])
print(0)
```

```
all 500 tests identical
```

On four test lines of 10^5 bars each (three random and one strictly increasing, whose answer is 2 500 050 000), the program runs in about 0.04 seconds on our machine, most of it spent reading the numbers.

4.7 From Histograms to Matrices

A favourite variation asks for the largest rectangle consisting only of 1s in a binary matrix with R rows and C columns. It is HISTOGRAM in disguise. For every row r and column c , let $up_r[c]$ be the number of consecutive 1s ending at row r in column c :

$$up_r[c] = \begin{cases} up_{r-1}[c] + 1 & \text{if cell } (r, c) \text{ is } 1 \\ 0 & \text{otherwise} \end{cases} \quad (4.4)$$

Every all-1 rectangle has a bottom row; for that row, the rectangle is a rectangle under the histogram up_r . So running the single pass of Section 4.5 on each row's histogram and taking the best answer solves the matrix problem in $O(R \cdot C)$ time. Equation (4.4) updates one row from the previous one in $O(C)$, so the whole algorithm is a few lines around Listing 4.3.

Other members of the family are worth knowing by name, because the national round likes them: the *trapping rain water* problem (how much water stays between bars after rain), the *sum of subarray minimums* (each element contributes h_i times the number of subarrays in which it is the minimum, which is again $(i - L_i)(R_i - i)$ from nearest smaller elements), and *next greater element* problems on circular arrays. Laaksonen [4] and Halim et al. [5] collect many of them.

The trapping-rain-water problem is a good second exercise in the same spirit. After rain, the water above bar i rises to the lower of the highest bar on its left and the highest bar on its right, so the water above bar i is $\min(\max L_i, \max R_i) - h_i$, where $\max L_i$ and $\max R_i$ include bar i itself. Two prefix maxima, one from each side, give the answer in $O(N)$ without any stack; a monotonic stack gives another $O(N)$ solution that fills the water layer by layer as each “valley” is closed. Seeing both solutions side by side is the best way to learn when a stack is necessary and when a simple prefix computation is enough.

KEY INSIGHT: When the answer for each element is decided by the *first* element to its left or right that breaks a condition (“the first shorter bar”, “the first higher price”), think of a monotonic stack. Elements that can no longer be anyone’s answer are popped, and each is popped only once.

4.8 Contest-Day Tactics

- **Recognize the keywords.** “Nearest smaller or greater”, “how long does it survive”, “maximal rectangle”, “span”, “visible”. Each points to a monotonic stack.
- **Decide strict or non-strict on purpose.** Write down what happens with equal values before coding, and test a case with ties, such as 3, 3.
- **Use a sentinel.** A height-0 bar at the end (or a very large one for a decreasing stack) removes the clean-up loop and its bugs.
- **Bank the $O(n^2)$ subtask.** Listing 4.4 earns the small subtask and becomes the stress-test reference.
- **Count stack operations, not loops,** when you estimate the running time.

4.9 Summary

- The best rectangle touches its lowest bar, so its area is Equation (4.1) for some bar, with L_i and R_i the nearest strictly smaller bars.

- A monotonic stack computes nearest smaller elements in one pass; its invariant keeps only positions that can still be someone's answer.
- Every index is pushed once and popped at most once, so the pass is $O(N)$ by Equation (4.2), despite the nested loops.
- HISTOGRAM computes each candidate area at pop time with the width of Equation (4.3), and a height-0 sentinel empties the stack at the end.
- Areas reach 10^{14} and must be `long long`.
- The largest all-1 rectangle in a matrix is HISTOGRAM on the row histograms of Equation (4.4).

4.10 Exercises

1. Trace **Listing 4.1** on the heights 3, 1, 3, 2, 2 and give L_i and R_i for each position.
2. What does **Listing 4.3** print for the input 5 1 2 3 4 5 followed by 0? Which pop produces the answer?
3. A friend changes the comparison in HISTOGRAM from `>` to `>=`. Is the program still correct? Explain.
4. Give a histogram of five bars on which the answer is achieved by *two* different rectangles.
5. Using Equation (4.4), write the row histograms of the matrix with rows 101, 111, 011, and find the largest all-1 rectangle.
6. Explain in two sentences why the stock-span stack of **Listing 4.2** performs at most $2N$ push and pop operations.

4.11 Solutions to the Exercises

1. $L = (-1, -1, 1, 1, 1)$ and $R = (1, 5, 3, 5, 5)$. For example position 2 (height 3) has position 1 (height 1) on the left and position 3 (height 2) on the right; position 4 (height 2) has no strictly smaller bar to its right, so $R_4 = 5$.
2. It prints 9. The bars are pushed in increasing order; the sentinel at $i = 5$ pops them from the top: (4, 5, width 1, area 5), (3, 4, width 2, area 8), (2, 3, width 3, area 9), (1, 2, width 4, area 8), (0, 1, width 5, area 5). The answer 9 comes from popping bar 2, the rectangle of height 3 over bars 2 to 4.
3. Yes. With `>=`, a bar is popped by an equal bar, and its width then stops early, but the *last* bar of a run of equal bars gets the full width, because its left neighbour in the stack is strictly lower. At least one bar of each maximal rectangle is still measured correctly, so the maximum is unchanged.
4. For example 2, 2, 0, 4, 1: the rectangle of height 2 over bars 0 and 1 and the single bar of height 4 both have area 4, and no rectangle is larger. The program prints 4 for it.
5. The histograms are (1,0,1), (2,1,2) and (0,2,3). Their best rectangles have areas 1, 3 and 4. The answer is 4: height 2 over the last two columns in rows 2 and 3.
6. Every day's pair is pushed exactly once, and a popped pair never returns, so it is popped at most once. The number of pops is therefore at most the number of pushes, N , and the total is at most $2N$.

References

- [1] Sphere Online Judge, "HISTOGRAM — Largest Rectangle in a Histogram." [Online]. Available: <https://www.spoj.com/problems/HISTOGRAM/> (accessed Sep. 2026).

- [2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.
- [3] R. E. Tarjan, "Amortized computational complexity," *SIAM Journal on Algebraic and Discrete Methods*, vol. 6, no. 2, pp. 306–318, 1985.
- [4] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.
- [5] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Lulu Press, 2020.

Chapter 5

Living Data Structures: Point Updates in Segment Trees — GSS3

The segment tree of Chapter 2 answered questions about an array that never changed. Real data changes: a ledger entry is corrected, a price is revised, a sensor reports a new reading. This chapter teaches the segment tree to live with change. You will learn the point update, which repairs the tree along a single root-to-leaf path in logarithmic time, prove why that path is enough, and see how the neutral element of an operation decides how a query handles ranges that do not overlap. Just as important, the chapter walks through a complete practice cycle as it would run on a contest day: read, design, implement, check the samples, stress-test, submit. The case study is GSS3, the problem of Chapter 2 with updates mixed into the questions.

5.1 Corrections in the Ledger

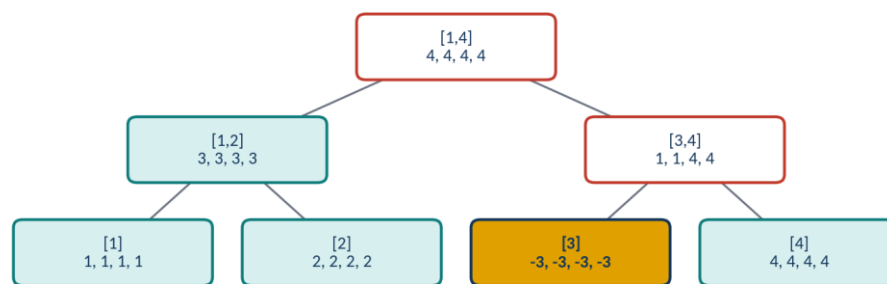
The fruit cooperative of Chapter 2 now has its app, and members use it every day to ask for the best stretch of days in any period. Then the treasurer discovers a mistake: the income of day 1 812 was typed as 3 500 instead of -3 500, because a refund had been recorded as a sale. She corrects the entry. From that moment, every answer that involves day 1 812 must use the corrected value.

The tree built in Chapter 2 does not know about the correction. Rebuilding it from scratch costs $O(N)$, and if corrections are as frequent as questions, N corrections and N questions cost $O(N^2)$ again, the slow solution we were trying to avoid. We need an update that costs about as much as a query: $O(\log N)$. That is exactly the SPOJ problem GSS3 [1]: the best-sum block inside a range, with operations that change one element in between the questions.

5.2 The Idea in Plain Words

When one element changes, which nodes of the tree become wrong? A node stores the answer for its range, so it becomes wrong only if its range contains the changed position. The ranges that contain a given position are exactly the nodes on the path from the root to that position's leaf: one node per level. Every other node describes a range the change does not touch, so its stored answer is still correct.

So the update is: go down to the leaf, replace it, and on the way back up recompute each node on the path from its two children, with the same `combine` used for building. The children are correct (one of them was just repaired, the other was never wrong), so each recomputed node is correct too. That is the whole idea, and **Figure 5.1** shows it on four elements: after the value at position 3 changes, only the leaf [3] and the two red nodes above it are recomputed.



after "0 3 -3": leaf [3] replaced (amber); red boxes recombined on the way up; all others untouched

Figure 5.1 The tree of 1, 2, -3, 4 after the update "0 3 -3": the amber leaf is replaced, the two red nodes on its path are recombined, and the other nodes keep their values.

5.2.1 How little changes

It is worth feeling how small the repair is. For $N = 50\,000$ the tree has about 10^5 nodes, and an update recomputes 16 of them, fewer than one in six thousand. The other nodes keep answers that are still true, and they keep being reused by every later question. This is the general principle behind efficient dynamic data structures: an operation should touch only the part of the stored information that the change actually invalidates. A structure that must be rebuilt after every change is only a static structure used repeatedly.

The same principle tells you when a segment tree will *not* work for updates. If changing one element could change the stored answer of nodes whose ranges do not contain it, the path repair would be wrong. That never happens when every node stores a function of its own range only, which is exactly what the closure recipe of Chapter 2 guarantees. So a node designed with the recipe supports point updates for free: the update of Section 5.4 needs no new idea, only the discipline to recombine on the way up.

5.3 Warm-Up: Range Sums with Updates

Start with the simplest living tree: range sums with point assignment. **Listing 5.1** supports two operations, `S x y` (print the sum of a_x to a_y) and `U x v` (set a_x to v).

Listing 5.1 `sum_update.cpp`: a segment tree for range sums with point assignment.

```
// Warm-up: segment tree for range sums with point assignment.
// "S x y": print a[x] + ... + a[y]; "U x v": set a[x] = v.
#include <cstdio>
#include <vector>
using namespace std;

int n;
vector<long long> t; // t[v] = sum of the range that node v covers

void update(int v, int lo, int hi, int pos, long long val) {
    if (lo == hi) { t[v] = val; return; }
    int mid = (lo + hi) / 2;
    if (pos <= mid) update(2 * v, lo, mid, pos, val);
    else update(2 * v + 1, mid + 1, hi, pos, val);
    t[v] = t[2 * v] + t[2 * v + 1]; // the only line that knows about "sum"
}

long long query(int v, int lo, int hi, int ql, int qr) {
    if (qr < lo || hi < ql) return 0; // no overlap: neutral value
    if (ql <= lo && hi <= qr) return t[v];
    int mid = (lo + hi) / 2;
    return query(2 * v, lo, mid, ql, qr) + query(2 * v + 1, mid + 1, hi, ql, qr);
}

int main() {
    int m;
    if (scanf("%d %d", &n, &m) != 2) return 0;
    t.assign(4 * n, 0);
    for (int i = 1; i <= n; i++) {
        long long v;
        if (scanf("%lld", &v) != 1) return 0;
        update(1, 1, n, i, v); // n updates also build the tree, in O(n log n)
    }
    while (m--) {
        char op;
        int x;
        long long y;
        if (scanf(" %c %d %lld", &op, &x, &y) != 3) return 0;
        if (op == 'U') update(1, 1, n, x, y);
        else printf("%lld\n", query(1, 1, n, x, (int)y));
    }
    return 0;
}
```

On the array 4, -1, 3, 0, 2, the sum of the whole array is 8; after setting a_2 to 10, the sum of the first three is 17; the fourth element alone is 0:

```
S 4
4 -1 3 0 2
S 1 5
U 2 10
S 1 3
S 4 4
```

Let us trace the query `S 1 3` after the update, on the tree over positions 1 to 5 (the root covers [1,5], its children [1,3] and [4,5]). The root overlaps the query only partly, so it asks both children. The left child [1,3] lies completely inside the query and returns its stored sum, $4 + 10 + 3 = 17$. The right child [4,5] does not overlap [1,3] at all and returns the neutral 0. The root adds them: 17. Two visits below the root were enough, and one of them cost nothing.

Three details of this warm-up prepare the case study.

The update's last line is the only line that knows the operation. `t[v] = t[2 * v] + t[2 * v + 1]` is the `combine` of sums. Change it to `max` and the same tree answers range maxima.

Building by updates. Instead of a separate `build`, the program inserts the N initial values with N updates. That costs $O(N \log N)$ instead of $O(N)$; for $N = 10^5$ the difference is a few milliseconds, and it removes a function that could contain a bug. In a contest, fewer functions is often the better trade.

A neutral element in the query. This query has a fourth case that the query of Chapter 2 did not need: if the node's range does not overlap the query at all, it returns 0. That works because 0 is the *neutral element* of addition: adding it changes nothing. Formally, an element e is neutral for an operation $\oplus$ when

$$e \oplus x = x \oplus e = x \quad \text{for every } x \quad (5.1)$$

For sums $e = 0$, for minima $e = +\infty$, for maxima $e = -\infty$, for products $e = 1$. With a neutral element, the query can simply ask both children every time and let the non-overlapping side return e . Without one, as for the four-field node of GSS1 and GSS3 (Section 2.6.5), the query must never visit a non-overlapping node, which is why it uses the three-case form.

5.4 The Point Update

5.4.1 The algorithm

`update(v, lo, hi, pos, val)` sets position `pos` to `val` inside node v 's range:

- If v is a leaf ($lo = hi$), replace its node by `makeNode(val)`.
- Otherwise descend into the child whose range contains `pos` (left if $pos \leq mid$, right otherwise), and after the recursive call returns, recompute v as `combine` of its two children.

5.4.2 Why it is correct

The tree is correct when every leaf describes its element and every inner node equals `combine` of its children; call this the *tree invariant*. Before an update the invariant holds. The update changes the leaf of `pos`, which is then correct for the new value. Going back up, each node on the path is recomputed from its children *after* its child on the path has been repaired, so it satisfies the invariant again. A node that is not on the path covers a range without `pos`, so neither its children nor its own value changed, and the invariant still holds for it. Hence the invariant holds everywhere after the update, and every later query is correct by the argument of Chapter 2.

The order matters: the recomputation must happen *after* the recursive call. Recomputing on the way down, before the child is repaired, copies the old value upward and leaves the tree silently wrong.

5.4.3 Cost

The path from the root to a leaf has one node per level, and the depth of the tree is $\lceil \log_2 N \rceil$, so an update performs

$$\text{combines per update} = \lceil \log_2 N \rceil \quad (5.2)$$

For $N = 50\,000$ that is 16 calls to `combine`. A sequence of M operations, updates and queries mixed, costs $O(N + M \log N)$ including the initial build.

5.4.4 A bottom-up alternative

The recursive update is the clearest form, but many experienced contestants also know a *bottom-up* segment tree, which has no recursion at all. Store the N leaves at indices N to $2N - 1$ of an array of size $2N$, and let node v (for $1 \leq v < N$) be the combination of nodes $2v$ and $2v + 1$. An update writes the leaf at index $N + pos$ and then walks upward with $v \leftarrow \lfloor v/2 \rfloor$, recombining each node, until it reaches the root. A query walks two indices, one from each end of the range, towards each other, collecting the nodes that fall completely inside. The bottom-up tree uses only $2N$ nodes and is roughly twice as fast, but its query collects the left and right parts in two separate accumulators, and for a non-commutative `combine` such as GSS's, the right-hand accumulator must be combined in reverse order. That detail is easy to get wrong under pressure, which is why this book uses the recursive form for the case studies. Learn the bottom-up form once you are comfortable with the recursive one, and test it with the same stress script.

5.5 A Practice Cycle That Imitates the Contest

A data-structure problem is where a clean working routine pays off most, because the bugs are silent: the program runs, prints plausible numbers, and is wrong on one test in fifty. **Table 5.1** is a two-hour practice routine for a problem like GSS3. Run it with a timer, alone or with a friend, and write down the time each stage actually took; after a few weeks those notes show you where your time goes.

Table 5.1 A two-hour practice routine with the rhythm of a contest day.

Minutes	Activity	Target
0–10	Read the problem; write the model and the node on paper	A one-sentence model and the combine formulas
10–45	Implement the full solution	A clean compile and the sample passing
45–70	Stress-test against an $O(N^2)$ brute force, fix bugs	Hundreds of random cases identical
70–85	Submit to the judge	An accepted verdict
85–120	Study a variant (range assignment, lazy propagation)	A plan for the next harder problem

Two rules make the routine honest. Do not open a finished solution before minute 70: the struggle of minutes 10 to 70 is where the learning happens. And allow yourself at most three hints, from a friend or a coach, which is roughly the amount of help a contestant gets from clarification requests in a real contest.

5.5.1 What to record while practising

The notes you take during the routine are worth more than the accepted verdict. For every stage, record three things: the minutes it took, the one mistake that cost the most time, and what would have caught that mistake earlier. After a few problems, patterns appear. Some contestants lose most of their time in minutes 10 to 45 because they start typing before the node is designed; the fix is to insist on writing the combine formulas on paper first. Others pass the sample quickly and then lose half an hour to a bug that the stress test would have found in a minute; the fix is to write the brute force and the generator *before* the fast solution, while the problem is still fresh. Others again are slowed by the compiler: they do not know how to read a template error message, or

they forget a flag. Each of these is a different weakness with a different cure, and only honest notes reveal which one is yours.

Keep the notes in the error notebook of Chapter 1. Read them before each practice session, and choose one habit to improve that day. Improvement in competitive programming comes less from solving more problems than from making fewer of the same mistakes.

5.6 HARD Case Study: GSS3

5.6.1 The problem in our own words

You are given $N \leq 50\,000$ integers $a_1, \dots, a_N$ of absolute value at most 10 000, and then $M \leq 50\,000$ operations, each a line of three integers. The operation `0 x y` sets a_x to y . The operation `1 x y` asks for the largest sum of a non-empty block $a_i + \dots + a_j$ with $x \leq i \leq j \leq y$, which must be printed on its own line. The input gives N , the array, M , and the operations.

5.6.2 Sample input and output

```
4
1 2 3 4
4
1 1 3
0 3 -3
1 2 4
1 3 3
```

On 1, 2, 3, 4, the best block inside [1,3] is the whole range: 6. Then a_3 becomes -3 . Inside [2,4] the values are 2, -3 , 4, and the best block is 4 alone ($2 - 3 + 4$ is only 3). Finally [3,3] contains only -3 , and because the block must be non-empty, the answer is -3 .

5.6.3 Design

The node, `makeNode`, `combine`, `build` and `query` are exactly those of GSS1, derived in Section 2.5. The only new function is `update` from Section 5.4. Operation 0 calls `update`; operation 1 calls `query` and prints the `ans` field.

What type should the fields have? The largest possible absolute value of a sum is

$$N \cdot \max|a_i| \leq 50\,000 \times 10\,000 = 5 \times 10^8 < 2^{31} - 1 \approx 2.15 \times 10^9 \quad (5.3)$$

and `suf + pre` inside `combine` adds two such values, at most 10^9 , still below the `int` limit. So `int` is safe here, by Equation (5.3), and the solution uses it. Doing this check explicitly, rather than guessing, is the habit that matters: GSS1's values were larger, and a problem with values up to 10^9 would need `long long`.

5.6.4 The full solution

Listing 5.2 is the complete program. Compared with the GSS1 solution of Chapter 2 (Listing 2.3), it adds `update` and the operation code, and it moves `query` before the markers, so that the part left to the reader in the public edition is `combine` together with `update`.

Listing 5.2 `GSS3.cpp`: the GSS1 tree with a point update; every operation costs $O(\log N)$.

```
// GSS3 (SPOJ): GSS1 plus point updates. "0 x y": a[x] = y;
// "1 x y": best non-empty subarray sum inside [x, y]. C++17.
#include <algorithm>
#include <cstdio>
using namespace std;

const int MAXN = 50001;
struct Node {
    int sum, pre, suf, ans; // |sum| <= 50000 * 10000, so int is enough
};
Node tree[4 * MAXN];
int a[MAXN], N;

Node makeNode(int v) {
    Node n;
    n.sum = n.pre = n.suf = n.ans = v;
    return n;
}

Node combine(const Node& l, const Node& r); // defined below

void build(int v, int lo, int hi) {
    if (lo == hi) {
        tree[v] = makeNode(a[lo]);
        return;
    }
    int mid = (lo + hi) / 2;
    build(2 * v, lo, mid);
    build(2 * v + 1, mid + 1, hi);
    tree[v] = combine(tree[2 * v], tree[2 * v + 1]);
}

Node query(int v, int lo, int hi, int ql, int qr) {
    if (ql <= lo && hi <= qr) return tree[v];
    int mid = (lo + hi) / 2;
    if (qr <= mid) return query(2 * v, lo, mid, ql, qr);
    if (ql > mid) return query(2 * v + 1, mid + 1, hi, ql, qr);
    return combine(query(2 * v, lo, mid, ql, qr),
        query(2 * v + 1, mid + 1, hi, ql, qr));
}
```

Listing 5.2 (Continued) GSS3.cpp: the GSS1 tree with a point update; every operation costs $O(\log N)$.

```
// <P50>
// combine: the four GSS1 formulas (sum, pre, suf, ans) for two neighbors.
// update: walk down to the leaf of pos, replace it with makeNode(val), and
// recombine every node on the way back up so all ancestors are fresh again.
// >>> YOUR TURN (Public Edition): write this part yourself, following the hint above. <<<
// </P50>

int main() {
    if (scanf("%d", &N) != 1) return 0;
    for (int i = 1; i <= N; i++)
        if (scanf("%d", &a[i]) != 1) return 0;
    build(1, 1, N);
    int M;
    if (scanf("%d", &M) != 1) return 0;
    while (M--) {
        int op, x, y;
        if (scanf("%d %d %d", &op, &x, &y) != 3) return 0;
        if (op == 0) update(1, 1, N, x, y);
        else printf("%d\n", query(1, 1, N, x, y).ans);
    }
    return 0;
}
```

5.6.5 Line by line

The node and the leaf. `Node` has four `int` fields justified by Equation (5.3). `makeNode(v)` sets all four to `v`, because the only non-empty block of one element is the element itself.

build and query. They are unchanged from Chapter 2. `build` fills the tree once in $O(N)$; `query` uses the three-case form, never visiting a node that does not overlap the query, because the four-field node has no neutral element.

combine. Between the markers `// <P50>` and `// </P50>`, `combine` restates the four formulas (2.5) to (2.8) of Chapter 2: sums add, the best prefix either stays in the left part or takes all of it and continues, the best suffix is the mirror image, and the best block is inside the left, inside the right, or crosses.

update. Also inside the markers, `update` is Section 5.4.1 in code. At the leaf it stores `makeNode(val)`. Otherwise it recurses into one child and then, *after* the call, recomputes `tree[v]` from both children. The recursion depth is at most 17 for $N = 50\,000$.

main. It reads N and the array, builds the tree, reads M , and processes each operation: `op == 0` updates, anything else queries and prints. Every `scanf` result is checked.

5.6.6 Traps

- **Forgetting to recombine.** An update that replaces the leaf but does not recompute the ancestors passes the sample only if no question after the update touches the changed position. The stress test finds it in the first few rounds.
- **Recombining before the recursion.** As explained in Section 5.4.2, this copies stale values upward.
- **Updating `a[x]` but not the tree.** The array `a` is only used by `build`; after that, the tree is the truth. Changing `a[x]` alone does nothing.
- **Mixing up the operation codes.** In GSS3, 0 means update and 1 means query. Read the statement twice.
- **Clamping at zero.** As in GSS1, `makeNode` must accept negative values, and no field may be clamped with `max(0, ...)`.

5.6.7 Stress test

Listing 5.3 keeps the array itself, applies updates directly, and answers questions by trying every block.

Listing 5.3 `brute.cpp`: updates on a plain array and every block of every range.

```
// Brute force for GSS3: change the array directly, scan every subarray.
#include <algorithm>
#include <climits>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n, m;
    cin >> n;
    vector<long long> a(n + 1);
    for (int i = 1; i <= n; i++) cin >> a[i];
    cin >> m;
    while (m--) {
        int op, x, y;
        cin >> op >> x >> y;
        if (op == 0) { a[x] = y; continue; }
        long long best = LLONG_MIN;
        for (int i = x; i <= y; i++) {
            long long s = 0;
            for (int j = i; j <= y; j++) best = max(best, s += a[j]);
        }
        cout << best << "\n";
    }
    return 0;
}
```

The generator **Listing 5.4** mixes updates and questions, about 40 percent updates, so that most questions follow at least one change.

Listing 5.4 `gen.py`: random GSS3 inputs mixing updates and questions.

```
#!/usr/bin/env python3
"""Random GSS3 input: python3 gen.py SEED"""
import random
import sys

rng = random.Random(int(sys.argv[1]) if len(sys.argv) > 1 else 1)
n = rng.randint(1, 12)
lim = rng.choice([3, 10, 10000])
print(n)
print(*[rng.randint(-lim, lim) for _ in range(n)])
m = rng.randint(1, 15)
print(m)
for _ in range(m):
    if rng.random() < 0.4:
        print(0, rng.randint(1, n), rng.randint(-lim, lim))
    else:
        x = rng.randint(1, n)
        print(1, x, rng.randint(x, n))
```

all 500 tests identical

On the largest input, $N = M = 50\,000$ with half updates and half questions, the program runs in about 0.04 seconds on our machine, as [Equation \(5.2\)](#) predicts.

5.6.8 A bug caught in the act

To see what the stress test buys you, imagine a solution in which `update` replaces the leaf correctly but forgets the line `tree[v] = combine(tree[2 * v], tree[2 * v + 1]);`. The official sample catches that one: after `0 3 -3`, the root still describes 1, 2, 3, 4, and the question `[2,4]` is answered from the stale node `[3,4]`, printing 7 instead of 4. A subtler mistake is to recombine only when the update goes *left*. Whether the sample catches it depends on where its single update

happens to fall, and reasoning about which samples a bug survives is a waste of contest time. Run the stress test instead. We planted exactly this bug in a copy of [Listing 5.2](#) and ran the script: it stopped at the very first seed and printed an input with an array of three numbers and eight operations. Shrinking that input by hand leads to something as small as the array 1, 2 with the update 0 2 5 and the question 1 1 2, where the correct answer is 6 and the buggy program prints 3. Tracing such a tiny input by hand takes a minute and points straight at the missing recombination.

The lesson is general. Samples are chosen by the problem setter to explain the statement, not to catch your particular bug. A generator explores the inputs you did not think of, and a brute force judges them, so the only thing left for you is to read a short failing case.

5.7 Beyond Point Updates

GSS3 changes one element at a time. Harder problems change a whole range at once: “add 5 to every element of $[l, r]$ ” or “set every element of $[l, r]$ to c ”. Doing that with point updates costs $O(N \log N)$ per operation, which is too slow. The standard remedy is *lazy propagation*: a node that is fully covered by the update records the pending change (a *tag*) and updates its own answer, without visiting its children; the tag is pushed down to the children only when a later operation needs to go below that node. For a range assignment of c on a node covering ℓ elements, the four GSS fields become

$$\text{sum} = c \ell, \quad \text{pre} = \text{suf} = \text{ans} = \max(c, c \ell) \quad (5.4)$$

because the best non-empty block of ℓ equal values c is the whole range when $c > 0$ and a single element otherwise. [Equation \(5.4\)](#) is the key formula of such a solution, and with it every operation stays $O(\log N)$.

A small example shows the saving. On eight elements, the assignment “set $[1, 8]$ to 2” touches only the root: it stores the tag “all 2” and, by [Equation \(5.4\)](#) with $\ell = 8$, the node (16, 16, 16, 16). A later question about $[3, 4]$ must go below the root, so the root first *pushes* its tag to its two children (each becomes (8, 8, 8, 8) with the tag “all 2”) and forgets it; the push continues only along the path the question actually takes. Nodes that no question ever visits keep their tags forever, and nobody pays for them.

Lazy propagation is the natural next step after this chapter; Laaksonen [2] and Halim et al. [3] develop it carefully. For the special case of prefix sums with point updates, the Fenwick tree [4] is shorter than any segment tree and worth knowing too.

The GSS problems on SPOJ form a well-known ladder: GSS1 (static, Chapter 2), GSS3 (point updates, this chapter), and later ones that add range updates, square roots of elements, or a condition that each value may be counted only once. Climbing that ladder is excellent training for the national round, and the four-field node you now know reappears in nearly every step.

5.8 Contest-Day Tactics

- **Reuse the static tree.** If a problem adds updates to a static range problem, keep `makeNode`, `combine` and `query`, and add the six-line `update`.
- **Check the numeric bound explicitly** with a computation like [Equation \(5.3\)](#) before choosing `int` or `long long`.

- **Generate updates in your stress test.** A generator without updates tests only the static part.
- **Decide on the neutral element.** If your operation has one (Equation (5.1)), a four-case query is simplest; if not, use the three-case query.
- **Follow the practice cycle of Table 5.1** in training, so that the rhythm is automatic on the day.

5.9 Summary

- Changing one element invalidates only the nodes on its root-to-leaf path, so a point update recomputes $\lceil \log_2 N \rceil$ nodes (Equation (5.2)) with the same `combine` used for building.
- Correctness rests on the tree invariant (every inner node equals `combine` of its children), restored bottom-up by the update.
- A neutral element (Equation (5.1)) lets a query visit non-overlapping nodes; the GSS node has none, so its query uses three cases.
- GSS3 is GSS1 plus the point update; its values fit in `int` by Equation (5.3).
- Range updates need lazy propagation, with formulas such as Equation (5.4) for each kind of tag.

5.10 Exercises

1. On the tree of Figure 5.1, which nodes are recomputed by the update “0 1 7”? Give their new four-field values.
2. What goes wrong if `update` recomputes `tree[v]` *before* the recursive call? Give a two-element example.
3. Write the `combine` and the neutral element for a tree that answers range minima together with the number of positions attaining the minimum.
4. The values of a GSS3-like problem may reach 10^6 in absolute value with $N \leq 10^5$. Is `int` still safe? Use a computation like Equation (5.3).
5. A friend’s GSS3 solution rebuilds the whole tree after every update. For $N = M = 50\,000$, estimate the number of `combine` calls and decide whether it passes a one-second limit.
6. Using Equation (5.4), give the node for a range of three elements all set to -2 , and for a range of three elements all set to 5 .
7. In a bottom-up segment tree (Section 5.4.4), a query for GSS3 collects nodes from the left end and from the right end of the range. In which order must the collected nodes be combined, and why?

5.11 Solutions to the Exercises

1. The path is the leaf $[1]$, node $[1,2]$ and the root. The leaf becomes $(7,7,7,7)$. Node $[1,2]$ combines $(7,7,7,7)$ with $[2] = (2,2,2,2)$: $(9,9,9,9)$. The root combines $(9,9,9,9)$ with $[3,4] = (1,1,4,4)$: $sum = 10, pre = \max(9,9 + 1) = 10, suf = \max(4,1 + 9) = 10, ans = \max(9,4,9 + 1) = 10$.
2. The node is recomputed from its old children, so it keeps the old answer, and the repair of the child is never propagated. For the array $1, 2$ and the update “0 1 5”, the root stays $(3,3,3,3)$ although the true answer for $[1,2]$ is now 7 .
3. The node is (minimum, count). `combine`: if the left minimum is smaller, take the left node; if the right is smaller, take the right node; if they are equal, keep the minimum and add the counts. The neutral element is $(+\infty, 0)$.
4. $10^5 \times 10^6 = 10^{11}$, far above 2.15×10^9 : `int` overflows, and all fields must be `long long`.

5. A rebuild costs about N combines; with about 25 000 updates that is 1.25×10^9 combines of four fields each, several seconds at best. It does not pass; the path update costs $25\,000 \times 16 = 4 \times 10^5$ combines.
6. For $c = -2$ and $\ell = 3$: $sum = -6$ and $pre = suf = ans = \max(-2, -6) = -2$. For $c = 5$: $sum = 15$ and the other three fields are $\max(5, 15) = 15$.
7. The nodes collected from the left end are met in left-to-right order, so they are combined as `leftAcc = combine(leftAcc, node)`. The nodes collected from the right end are met in right-to-left order, so they must be combined as `rightAcc = combine(node, rightAcc)`, and the answer is `combine(leftAcc, rightAcc)`. The order matters because `combine` is not commutative: *pre* and *suf* describe the left and right edges, and swapping the arguments swaps the edges.

References

- [1] Sphere Online Judge, “GSS3 — Can you answer these queries III.” [Online]. Available: <https://www.spoj.com/problems/GSS3/> (accessed Sep. 2026).
- [2] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.
- [3] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Lulu Press, 2020.
- [4] P. M. Fenwick, “A new data structure for cumulative frequency tables,” *Software: Practice and Experience*, vol. 24, no. 3, pp. 327–336, 1994.

Chapter 6

State-Space Graphs and Dial's Algorithm — CHIPSLL

Many shortest-path problems at the national olympiad never use the word “graph”. They talk about a maze, a board game, a robot with a battery, two jugs of water or a hero fighting through a map full of monsters. The first skill this chapter builds is *seeing* the hidden graph: deciding what a node is, what an edge is and what each edge costs. The second skill is choosing the right tool once the graph is visible. Plain breadth-first search handles edges of cost 1, a double-ended queue handles costs 0 and 1, a ring of buckets (Dial's algorithm) handles small integer costs, and only then does the familiar heap-based Dijkstra algorithm earn its place. By the end of the chapter you will be able to model a puzzle as a state space, pick the fastest correct algorithm from the shape of the edge weights, and implement Dial's algorithm on a grid with four million cells, which is exactly what the SPOJ problem CHIPSLL demands [1].

6.1 A Hero, a Map and a Stopwatch

Imagine a small video game. The map is a rectangle of squares. Our hero, Chip, stands on one square; his friend Melinda waits on another. Some squares are empty. Others hold a puddle of water, a sleeping monster or a locked door, and each of those obstacles has a *power*, a whole number that says how many extra seconds it takes to clear it away. Walking into an empty square takes one second. Walking into a square with an obstacle takes one second plus the obstacle's power. Chip may move up, down, left or right. How quickly can he reach Melinda?

A child playing the game would look for a route that avoids the strongest obstacles, and would happily walk a few extra squares to go around a big monster. That instinct is correct, and it is exactly what a shortest-path algorithm formalizes. The difficulty is size. In CHIPSLL the map can be 2000 by 2000 squares, so there are four million squares and about sixteen million possible single steps, and a test file may contain ten such maps. The time limit leaves no room for a careless algorithm, and the memory limit leaves no room for careless data structures.

The same shape of question appears everywhere outside games. A courier on a motorbike crossing a city where each street segment has a different number of minutes of traffic; a farmer moving a water pipe across fields where some plots are harder to dig; a robot vacuum that pays a small energy price for every carpet edge it climbs. In each case the “cost” of a step is a small whole number, and that small detail is what this chapter exploits.

6.2 Seeing the Hidden Graph

6.2.1 States, actions and costs

A *graph* is a set of nodes joined by edges. A *weighted* graph also gives each edge a number, its *weight* or *cost*. A *shortest path* from a node s to a node t is a sequence of edges leading from s to t whose total weight is as small as possible; that total is called the *distance* $d(s, t)$.

DEFINITION. A *state space* is a graph whose nodes are the possible situations (states) of a puzzle or process and whose edges are the legal actions that change one state into another. The weight of an edge is the price of that action: time, moves, energy or money.

In a grid maze the state is simply the square you stand on. In a puzzle with keys and doors the state must also remember which keys you hold, so one square becomes several states. In the classic two-jug puzzle the state is the pair (water in jug 1, water in jug 2), and there is no “map” at all. The discipline that wins contests is to write four answers on paper before writing any code:

1. What is a state? (Which information must I remember to know what can happen next?)
2. How many states are there? (Multiply the ranges of the pieces.)
3. What are the actions from a state? (These are the edges.)
4. What does each action cost? (These are the weights, and their *shape* chooses the algorithm.)

For CHIPSLL the answers are short. A state is a square (r, c) , so there are at most $2000 \cdot 2000 = 4\,000\,000$ states. From each square there are at most four actions. The cost of stepping into square v is

$$w(v) = 1 + \text{power}(v), \quad 0 \leq \text{power}(v) \leq 62 \quad (6.1)$$

so every weight lies between 1 and 63. Notice from Equation (6.1) that the weight depends only on the square you *enter*, not on the square you leave. That is typical of grid problems, and it lets us store one small number per square instead of one per edge.

6.2.2 The weapon ladder

Once the state space is clear, the weights decide the algorithm. **Table 6.1** lists the choices from the simplest to the most general. Each row can solve every problem in the rows above it, but more slowly; a contestant who always reaches for the last row wastes time the jury has deliberately left for the better idea.

Table 6.1 The shortest-path ladder: the shape of the weights chooses the algorithm; D is the largest distance found.

Weights on the edges	Algorithm	Running time
all equal to 1	breadth-first search (BFS) with a queue	$O(V + E)$
only 0 or 1	0-1 BFS with a double-ended queue	$O(V + E)$
small integers $1..C$	Dial's algorithm, $C + 1$ circular buckets	$O(V + E + D)$
any non-negative	Dijkstra's algorithm with a binary heap	$O(E \log V)$
some negative	Bellman-Ford (or SPFA in practice)	$O(VE)$

Here V is the number of nodes and E the number of edges. For CHIPSLL, $V = 4 \cdot 10^6$ and $E \approx 1.6 \cdot 10^7$ per map. A heap performs about $\log_2 V \approx 22$ elementary steps per operation, with poor memory locality; buckets perform a constant number. The difference decides between *Accepted* and *Time Limit Exceeded*. The standard competitive-programming handbooks teach the same ladder, and are worth reading alongside this chapter [2], [3].

6.2.3 Why Dijkstra works, in one paragraph

Dijkstra's algorithm [4], [5] keeps a tentative distance $d[v]$ for every node, initially ∞ except $d[s] = 0$. It repeatedly takes the unfinished node u with the smallest tentative distance, declares it finished and *relaxes* its edges:

$$d[v] \leftarrow \min(d[v], d[u] + w(u, v)) \quad (6.2)$$

Why is the smallest tentative distance already final? Suppose some other route reached u more cheaply. That route leaves the finished region somewhere, at a first unfinished node x ; the part of the route up to x already costs at least $d[x] \geq d[u]$, and the remaining edges have non-negative weights, so the route costs at least $d[u]$. Contradiction. The whole algorithm is correct as long as we can repeatedly find “the unfinished node with the smallest tentative distance”. A heap is one way to find it. Dial’s idea is a cheaper way when the weights are small integers.

6.2.4 Dial’s algorithm: a ring of buckets

In 1969 Robert Dial observed that when all weights are integers between 1 and C , the tentative distances that are still waiting to be processed can never be spread over more than $C + 1$ consecutive values [6]. Suppose the algorithm is currently finishing nodes at distance cur . Every waiting value was produced by relaxing an edge from some finished node u with $d[u] \leq cur$, so

$$cur \leq d[v] \leq cur + C \quad \text{for every waiting node } v \quad (6.3)$$

The window in Equation (6.3) has $C + 1$ values. If we keep an array of $C + 1$ lists, the *buckets*, and put a node with tentative distance d into bucket $d \bmod (C + 1)$, then two different waiting distances can never fall into the same bucket. To find the next node to finish we simply look at bucket $cur \bmod (C + 1)$; if it is empty we advance cur by one and look at the next bucket, wrapping around the ring. **Pseudocode 6.1** writes the procedure out.

Pseudocode 6.1 Dial’s algorithm with $C + 1$ circular buckets; line 6 skips copies left behind when a node’s distance improved.

```
DIAL(G, s, C)
1 for each node v: d[v] = INF
2 d[s] = 0; put s into bucket[0]; cur = 0
3 while some bucket is non-empty
4     while bucket[cur mod (C+1)] is empty: cur = cur + 1
5     take any node u out of bucket[cur mod (C+1)]
6     if d[u] != cur: continue // stale copy, ignore it
7     for each edge (u, v) with weight w
8         if cur + w < d[v]
9             d[v] = cur + w
10            put v into bucket[(cur + w) mod (C+1)]
```

Line 6 needs a word. When a node’s tentative distance improves from 40 to 35, we do not search bucket 40 to delete the old copy; that would be slow. We simply add a new copy to bucket 35 and leave the old one where it is. When the old copy is finally taken out, its recorded distance no longer matches $d[v]$ and we throw it away. This *lazy deletion* is the same trick most contestants already use with heap-based Dijkstra.

The running time follows from counting. Every successful relaxation adds one copy to some bucket, so there are at most E copies in total, each taken out once. The pointer cur only moves forward and stops at the largest distance D , so empty-bucket checks cost $O(D)$ in total. Together that is $O(V + E + D)$, and since $D \leq C(V - 1)$ the classic bound is $O(E + VC)$. For a grid the path distances are far below that worst case, and the practical behavior is linear.

Figure 6.1 shows the distances Dial computes on a small 5×6 map. The number in each square is its entry cost w , and the smaller label is its distance from the start. The shaded squares form

one cheapest route of total cost 17; notice that it happily takes a detour down the left edge to avoid the 9 and the 8 near the start.

start (cost ignored)

1 d=0	9 d=9	3 d=12	1 d=13	7 d=20	2 d=22
2 d=2	8 d=10	1 d=10	6 d=16	1 d=17	5 d=22
4 d=6	1 d=7	2 d=9	9 d=18	3 d=20	1 d=20
3 d=9	7 d=14	1 d=10	1 d=11	8 d=19	2 d=19
5 d=14	2 d=16	6 d=16	4 d=15	1 d=16	1 d=17

target

Figure 6.1 A 5×6 map with entry costs (large numbers) and the distances d computed by Dial's algorithm; the shaded squares form a cheapest route of cost 17.

KEY INSIGHT: Dial's algorithm *is* Dijkstra's algorithm. Only the data structure that finds “the smallest tentative distance” changes: a ring of $C + 1$ lists instead of a heap. Everything you know about correctness, stale entries and early stopping carries over unchanged.

6.2.5 A state space without a map

To see that the method does not depend on a picture, consider the old two-jug puzzle. A 5-liter jug and a 3-liter jug stand next to a tap; we want exactly 4 liters in the large jug. The state is the pair (a, b) , the liters in the large and small jug, with $0 \leq a \leq 5$ and $0 \leq b \leq 3$, so there are $6 \cdot 4 = 24$ states. The actions are: fill a jug from the tap, empty a jug onto the ground, or pour one jug into the other until the first is empty or the second is full. If every action costs one move, BFS from $(0, 0)$ finds the fewest moves; the answer is six, through $(5, 0)$, $(2, 3)$, $(2, 0)$, $(0, 2)$, $(5, 2)$ and $(4, 3)$.

Change one sentence of the story and the weights change with it. If filling from the tap costs the liters of water used and pouring or emptying costs nothing, the weights become the small integers 0 to 5, and the right tool moves from BFS to Dial's algorithm with six buckets. Nothing else in the program changes: the states and the actions are the same, only the costs are different. That is the whole point of separating the four questions of Section 6.2.1. In a contest, a problem that looks unfamiliar is very often an ordinary shortest-path problem over states you have not yet written down, and the ten minutes spent writing them down are the most valuable ten minutes of the problem.

6.3 Warm-Up Problems

Before the case study, two small programs exercise the lower rungs of the ladder. Each is complete and runs as it is.

6.3.1 A knight on a board with forbidden squares

A chess knight starts on a1 and wants to reach h8; some squares are forbidden. Every move costs 1, so plain BFS is the right tool. The state is the knight's square (64 states) and the actions are the eight knight jumps. **Listing 6.1** runs three scenarios.

Listing 6.1 `knight.cpp`: breadth-first search over the 64 knight states, with forbidden squares.

```
// State-space BFS: fewest knight moves from a1 to h8 avoiding forbidden squares.
#include <iostream>
#include <queue>
#include <string>
#include <vector>
using namespace std;
int knightDist(const vector<string>& banned) {
    bool bad[8][8] = {};
    for (auto& s : banned) bad[s[0] - 'a'][s[1] - '1'] = true;
    int dist[8][8];
    for (auto& row : dist)
        for (int& x : row) x = -1;
    int dx[8] = {1, 2, 2, 1, -1, -2, -2, -1}, dy[8] = {2, 1, -1, -2, -2, -1, 1, 2};
    queue<pair<int, int>> q;
    dist[0][0] = 0;
    q.push({0, 0});
    while (!q.empty()) {
        auto [x, y] = q.front();
        q.pop();
        for (int k = 0; k < 8; k++) {
            int nx = x + dx[k], ny = y + dy[k];
            if (nx < 0 || nx > 7 || ny < 0 || ny > 7 || bad[nx][ny]) continue;
            if (dist[nx][ny] != -1) continue; // first visit = shortest in BFS
            dist[nx][ny] = dist[x][y] + 1;
            q.push({nx, ny});
        }
    }
    return dist[7][7]; // -1 if h8 was never reached
}
int main() {
    cout << "no square banned:      " << knightDist({}) << " moves\n";
    cout << "b3 and f7 banned:      " << knightDist({"b3", "f7"}) << " moves\n";
    cout << "f7 and g6 banned:      " << knightDist({"f7", "g6"}) << "\n";
}
```

```
no square banned:      6 moves
b3 and f7 banned:      6 moves
f7 and g6 banned:      -1
```

The first two answers show that a knight needs six moves from corner to corner, and that banning b3 and f7 leaves other six-move routes open. The third answer is a small lesson in modeling: f7 and g6 are the only two squares from which a knight can jump onto h8, so banning both makes h8 unreachable. BFS reports this correctly because `-1` was the initial value and h8 was never discovered. In BFS the first time a node is discovered is also the moment its distance is final, which is why the program needs no “stale” check at all.

6.3.2 Breaking the fewest walls: 0-1 BFS

Now each square of a small map is either free (.) or a wall (#). Stepping into a free square costs 0 and stepping into a wall (breaking it) costs 1. With weights 0 and 1 we can keep the nodes in a double-ended queue: a node reached by a 0-edge goes to the *front*, because it has the same distance as the current node, and a node reached by a 1-edge goes to the *back*. The deque then always holds at most two distinct distances in sorted order, which is Dial's algorithm with only two buckets. **Listing 6.2** prints the answer and the whole distance table.

Listing 6.2 `zero_one_bfs.cpp`: 0-1 BFS with a double-ended queue on a map of free squares and walls.

```
// 0-1 BFS: fewest walls to break from the top-left to the bottom-right cell.
// Stepping into '.' costs 0, stepping into '#' (breaking it) costs 1.
#include <deque>
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main() {
    vector<string> g = {"..#....",
                       "###.##.",
                       "...#..#",
                       ".#.#.#.",
                       ".#...#."};
    int R = g.size(), C = g[0].size();
    vector<vector<int>>> d(R, vector<int>(C, 1 << 30));
    deque<pair<int, int>> dq;
    d[0][0] = 0;
    dq.push_back({0, 0});
    int dr[4] = {-1, 1, 0, 0}, dc[4] = {0, 0, -1, 1};
    while (!dq.empty()) {
        auto [r, c] = dq.front();
        dq.pop_front();
        for (int k = 0; k < 4; k++) {
            int nr = r + dr[k], nc = c + dc[k];
            if (nr < 0 || nr >= R || nc < 0 || nc >= C) continue;
            int w = (g[nr][nc] == '#'); // edge weight 0 or 1
            if (d[r][c] + w < d[nr][nc]) {
                d[nr][nc] = d[r][c] + w;
                if (w == 0) dq.push_front({nr, nc}); // weight 0: same layer
                else dq.push_back({nr, nc});        // weight 1: next layer
            }
        }
    }
    cout << "walls to break: " << d[R - 1][C - 1] << "\n";
    for (int r = 0; r < R; r++) {
        for (int c = 0; c < C; c++) cout << d[r][c] << (c + 1 < C ? ' ' : '\n');
    }
}
```

```
walls to break: 2
0 0 1 1 1 1 1
1 1 2 1 2 2 1
1 1 1 2 1 1 2
1 2 1 2 1 2 2
1 2 1 1 1 2 2
```

Two walls are enough. Reading the table row by row is a good habit: every free square next to a square with distance k has distance at most k , and every wall next to it has distance at most $k + 1$. If a hand check of that rule fails anywhere, the implementation is wrong.

6.4 Case Study: CHIPSLL

6.4.1 The problem in our own words

The problem is SPOJ CHIPSLL [1]. The input starts with the number of test cases T (at most 10). Each test case gives six integers N, M, C_x, C_y, M_x, M_y and then a seed. The map has N rows and M columns, $N, M \leq 2000$. Chip starts on square (C_x, C_y) and Melinda waits on square (M_x, M_y) ; rows and columns are numbered from 1.

The map is *not* given in the input. It is produced by a pseudorandom generator that starts from the seed. Squares are visited row by row, left to right, skipping Chip's and Melinda's squares. For every other square the seed is updated as

$$seed \leftarrow (seed \cdot 1\,000\,003 + 10\,007) \bmod 1\,000\,000\,009 \quad (6.4)$$

and the square's power becomes $p = seed \bmod 63$. A power of 0 means an empty square; 1 to 26 means water, 27 to 52 a monster, 53 to 62 a door. What kind of obstacle it is does not matter for the time: entering the square always costs $1 + p$ seconds, as in Equation (6.1). Chip's and Melinda's squares cost 1. The output for test case t is a line **Case t: X** where X is the smallest total time.

Two warnings follow from this description. First, the generator must be reproduced *exactly*: the same constants, the same order of squares, the same skipped squares. A single wrong digit in Equation (6.4) produces a completely different map and a wrong answer on every test. Second, the product $seed \cdot 1\,000\,003$ reaches about 10^{15} , so it must be computed in 64-bit arithmetic (**long long**).

6.4.2 A sample worked by hand

Take the tiny input

```
1
2 3 1 1 2 3
7
```

The map has two rows and three columns; Chip is at (1,1) and Melinda at (2,3). The generator visits (1,2), (1,3), (2,1) and (2,2), skipping the two special squares. Starting from seed 7, Equation (6.4) produces 7010028, 48977001, 147500217 and 658183158, whose remainders modulo 63 are 18, 45, 18 and 45. The entry costs are therefore

```
1  19 46
19 46  1
```

Every route from the top-left to the bottom-right corner that uses three steps crosses one 19, one 46 and Melinda's 1, so the answer is $19 + 46 + 1 = 66$. The program agrees:

```
Case 1: 66
```

The file **sample.txt** in the chapter folder holds two larger maps (3×4 and 5×5), for which the program prints **Case 1: 125** and **Case 2: 160**; the stress test in Section 6.5 checks such answers automatically.

6.4.3 Designing the solution

With the model from Section 6.2 the design is almost mechanical.

1. **Generate the costs.** Store $w(v) = 1 + p$ for every square in an array of `unsigned char`: 63 fits in one byte, and four million bytes are cheap.
2. **Flatten the grid.** Give square (i, j) the index $i \cdot 2021 + j$. A one-dimensional array is faster than a vector of vectors: one allocation, no pointer chasing, and neighbors are simple offsets (± 1 for left and right, ± 2021 for up and down).
3. **Run Dial's algorithm** with $W = 64$ buckets. Since $C = 63$, the window Equation (6.3) has exactly 64 values. Stop as soon as Melinda's square is taken out of a bucket with a current distance: from that moment its distance is final.
4. **Print Case t: X.**

It pays to add up the memory before writing the code. The cost array needs $2021 \cdot 2021 \approx 4.1$ million bytes, about 4 MB. The distance array holds the same number of `int` values, about 16 MB. The buckets are the only part whose size depends on the map: every successful relaxation adds one entry of four bytes, and although a square can be improved several times before it is finished, on these maps the buckets stay at a few million entries at most, and the memory of an emptied bucket is reused as the ring turns. Altogether the program stays in the region of 30 to 50 MB, comfortably inside the usual limits. The same estimate for a heap-based version gives a similar distance array but a heap of pairs (distance, square), eight bytes per entry, and a heap never shrinks its storage while the program runs; that is one more reason the bucket version is the safer choice here.

To see how large a speed difference one can expect, picture the work of each version. The bucket version does a few simple operations per relaxation: append to a vector and later take the entry back. The heap version must keep its whole content ordered on every insertion and removal, swapping elements along a path as tall as the heap. Over millions of relaxations, that difference shows.

The number 2021 as the row stride deserves a comment. Using a stride slightly larger than the largest M lets rows 0 and $N + 1$ and columns 0 and $M + 1$ exist as unused padding, and it keeps the index formula identical for every test case, whatever its M .

6.4.4 The complete solution

Listing 6.3 is the full program, recorded as *Accepted* on SPOJ. The lines between `// <P50>` and `// </P50>` are the second half of the core: the relaxation of the four neighbors.

Listing 6.3 `CHIPSLL.cpp`: Dial's algorithm with 64 circular buckets on a flat grid.

```
#include <stdio>
#include <cstring>
#include <vector>
using namespace std;
static const int MAXN = 2021;
static const int MAXCELL = MAXN * MAXN;
static const int W = 64;
static const int INF = 0x3f3f3f3f;
static unsigned char pwv[MAXCELL]; // edge cost to enter cell (1 + power)
static int dist_flat[MAXCELL];
static long long seed_g;
static void update_seed(){
    seed_g = (seed_g * 1000003LL + 10007LL) % 1000000009LL;
}
static void gen_pwv(int N, int M, int Cx, int Cy, int Mx, int My){
    for(int i=1;i<=N;++i)
        for(int j=1;j<=M;++j){
            int id=i*MAXN+j;
            if((i==Cx&&j==Cy)||((i==Mx&&j==My))){pwv[id]=1;continue;}
            update_seed();
            int p=seed_g%63;
            // power(cell) == p exactly (water p<=26, monster p<=52, door p<=62)
            pwv[id]=(unsigned char)(1+p);
        }
}
static vector<int> bkt[W];
static int solve(int N, int M, int sr, int sc, int tr, int tc){
    int start=sr*MAXN+sc, target=tr*MAXN+tc;
    for(int i=1;i<=N;i++) memset(dist_flat+i*MAXN+1, 0x3f, M*sizeof(int));
    for(int i=0;i<W;i++) bkt[i].clear();
    dist_flat[start]=0;
    bkt[0].push_back(start);
    int cur_d=0;
    while(true){
        bool found=false;
        for(int s=0;s<W;s++){
            if(!bkt[(cur_d+s)%W].empty()){cur_d+=s;found=true;break;}
        }
        if(!found) break;
        int b=cur_d%W;
        while(!bkt[b].empty()){
            int id=bkt[b].back(); bkt[b].pop_back();
            if(dist_flat[id]!=cur_d) continue;
            if(id==target) return cur_d;
            // <P50>
            // Relax the four neighbors of id: nd = cur_d + entry cost of the
```

Listing 6.3 (Continued) *CHIPSSL.cpp: Dial's algorithm with 64 circular buckets on a flat grid.*

```

// neighbor; if nd improves it, store it and push into bucket nd%W.
// >>> YOUR TURN (Public Edition): write this part yourself, following the hint
above. <<<
// </P50>
}
cur_d++;
}
return dist_flat[target];
}
int main(){
    int T; if(scanf("%d",&T)!=1) return 0;
    for(int t=1;t<=T;t++){
        int N,M,Cx,Cy,Mx,My;
        if(scanf("%d%d%d%d%d",&N,&M,&Cx,&Cy,&Mx,&My)!=6) return 0;
        if(scanf("%lld",&seed_g)!=1) return 0;
        gen_pwv(N,M,Cx,Cy,Mx,My);
        printf("Case %d: %d\n",t,solve(N,M,Cx,Cy,Mx,My));
    }
}

```

6.4.5 Line by line

Constants and arrays. `MAXN = 2021` is the row stride and `MAXCELL` the size of the flat arrays. `W = 64` is the number of buckets. `pwv` holds the entry cost of every square and `dist_flat` the tentative distances; both are `static` globals, so they live in the program's data segment rather than on the stack, which would overflow at 16 MB.

update_seed and gen_pwv. These implement Equation (6.4) literally. The loop visits rows $1..N$ and columns $1..M$ in order, sets the cost of the two special squares to 1 *without* advancing the seed, and otherwise advances the seed once and stores $1 + (\text{seed} \bmod 63)$.

Initialization in solve. `memset` with the byte `0x3f` fills each `int` with `0x3f3f3f3f`, about $1.06 \cdot 10^9$, a convenient “infinity” that still leaves room to add a weight without overflow. Only the N used rows are reset, and only their M used columns; resetting all four million cells for a 10×10 test would waste time over ten test cases. The buckets are cleared, the start gets distance 0 and goes into bucket 0.

Finding the next non-empty bucket. The inner `for` over `s` looks at most W buckets ahead, starting from `cur_d`. If all 64 are empty, no node is waiting and the loop ends (with CHIPSSL's connected grids this happens only after the target is found). Otherwise `cur_d` jumps directly to the first non-empty distance.

Processing a bucket. Nodes are popped from the *back* of the vector: inside one bucket every node has the same distance, so the order does not matter, and `pop_back` is the cheapest operation a vector has. The check `dist_flat[id] != cur_d` discards stale copies (line 6 of Pseudocode 6.1). The check `id == target` stops early.

Relaxation (the P50 block). The row and column are recovered from the flat index with a division and a remainder. The lambda `relax` applies Equation (6.2) to one neighbor, and four guarded calls cover up, down, left and right. The guards `r>1`, `r<N`, `c>1`, `c<M` keep Chip inside the map.

main. It reads the header of each test, the seed into the global `seed_g`, generates the map and prints the answer in the required format. Every `scanf` result is checked, which keeps the compiler quiet under `-Wall -Wextra` and stops cleanly at the end of the input.

6.4.6 Traps

PITFALL: Using $W = 63$. The weights go up to 63, so the window Equation (6.3) holds 64 distinct values. With 63 buckets, distance d and $d + 63$ share a slot, a node can be processed at the wrong time, and answers become wrong only on some maps, which is the worst kind of bug.

- **Advancing the seed on the special squares.** The generator skips them. Advancing the seed there shifts every later square.
- **32-bit seed arithmetic.** `seed_g * 1000003` overflows `int`; the program uses `long long` and the `LL` suffix.
- **Forgetting stale entries.** Without the `dist_flat[id] != cur_d` check the program is still correct but may relax the same node many times and run out of time.
- **A vector of vectors for the grid.** Four million small vectors cost memory for their headers and time for their allocations; the flat array avoids both.
- **Resetting the whole array per test.** Correct but slow when many small tests follow a large one; reset only what the test uses.

6.5 Stress Testing Against a Brute Force

A program for a large problem is trustworthy only after it has been compared with a slow but obviously correct program on many small random inputs. Here the slow program, **Listing 6.4**, uses the same generator but the textbook heap-based Dijkstra on a vector of vectors. It shares no data structure with the fast solution, so a bug in one is unlikely to be repeated in the other.

Listing 6.4 `brute.cpp`: the reference solution with a binary heap.

```
// Reference for CHIPSLL: same grid generator, textbook heap Dijkstra.
#include <stdio>
#include <queue>
#include <vector>
using namespace std;
long long seed_g;
int main() {
    int T;
    if (scanf("%d", &T) != 1) return 0;
    for (int t = 1; t <= T; t++) {
        int N, M, Cx, Cy, Mx, My;
        if (scanf("%d%d%d%d%d%d%lld", &N, &M, &Cx, &Cy, &Mx, &My, &seed_g) != 7)
            return 0;
        vector<vector<int>> cost(N + 1, vector<int>(M + 1, 1));
        for (int i = 1; i <= N; i++)
            for (int j = 1; j <= M; j++) {
                if ((i == Cx && j == Cy) || (i == Mx && j == My)) continue;
                seed_g = (seed_g * 1000003LL + 10007LL) % 1000000009LL;
                cost[i][j] = 1 + (int)(seed_g % 63);
            }
        const int INF = 1 << 30;
        vector<vector<int>> d(N + 1, vector<int>(M + 1, INF));
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
        d[Cx][Cy] = 0;
        pq.push({0, Cx * (M + 1) + Cy});
        int dr[4] = {-1, 1, 0, 0}, dc[4] = {0, 0, -1, 1};
        while (!pq.empty()) {
            auto [du, id] = pq.top();
            pq.pop();
            int r = id / (M + 1), c = id % (M + 1);
            if (du != d[r][c]) continue;
            for (int k = 0; k < 4; k++) {
                int nr = r + dr[k], nc = c + dc[k];
                if (nr < 1 || nr > N || nc < 1 || nc > M) continue;
                if (du + cost[nr][nc] < d[nr][nc]) {
                    d[nr][nc] = du + cost[nr][nc];
                    pq.push({d[nr][nc], nr * (M + 1) + nc});
                }
            }
        }
        printf("Case %d: %d\n", t, d[Mx][My]);
    }
}
```

The generator **Listing 6.5** writes up to four random maps of at most 9×9 squares, with random start, target and seed. Because it takes the random seed from the command line, every failing case can be reproduced by rerunning the same number.

Listing 6.5 `gen.py`: random small inputs for CHIPSLL.

```
# Random small CHIPSLL inputs: python3 gen.py SEED
import random, sys
random.seed(int(sys.argv[1]))
T = random.randint(1, 4)
print(T)
for _ in range(T):
    n, m = random.randint(1, 9), random.randint(1, 9)
    print(n, m, random.randint(1, n), random.randint(1, m),
          random.randint(1, n), random.randint(1, m))
    print(random.randint(0, 10**9))
```

Finally, **Listing 6.6** compiles both programs with warnings enabled, runs them on 300 generated inputs and stops at the first difference, printing the input that caused it. The same script, with only the problem name changed, is used in every chapter of this book.

Listing 6.6 `stress.sh`: compile, generate, compare, stop at the first mismatch.

```
#!/usr/bin/env bash
# Stress test: CHIPSLL against brute.cpp on random small inputs.
# usage: bash stress.sh [ROUNDS] (default 300)
set -e
cd "$(dirname "$0")"
FLAGS="-std=c++17 -O2 -Wall -Wextra"
g++ $FLAGS -o sol CHIPSLL.cpp
g++ $FLAGS -o brute brute.cpp
for i in $(seq 1 "${1:-300}"); do
    python3 gen.py "$i" > in.txt
    ./sol < in.txt > out_sol.txt
    ./brute < in.txt > out_brute.txt
    if ! cmp -s out_sol.txt out_brute.txt; then
        echo "MISMATCH on seed $i"; cat in.txt; exit 1
    fi
done
echo "all ${1:-300} random tests passed"
```

```
all 300 random tests passed
```

Small tests check correctness; they say nothing about speed. So we also timed the largest legal input: ten maps of 2000×2000 . The whole file took about 1.1 seconds on our machine, roughly 0.11 seconds per map including generation. On one such map the heap version took about 0.66 seconds against 0.19 seconds for Dial's algorithm, a factor of three and a half that comes entirely from replacing the heap by buckets.

6.6 Contest-Day Tactics

- **Classify the weights first.** Before choosing an algorithm, write the smallest and largest edge weight in the margin of your scratch paper. The ladder in **Table 6.1** then chooses for you.
- **Copy generators character by character.** When a problem generates its own input, type the generator from the statement slowly and then print the first few generated values for a tiny case and compare them by hand, as in Section 6.4.2.
- **Estimate memory before coding.** Four million `int` values are 16 MB; four million `unsigned char` values are 4 MB. A map of `pair<int,int>` per cell would need several times more. Know these numbers before you choose types.
- **Keep a heap-Dijkstra template ready.** It is the brute force for every faster shortest-path method, and writing it from memory in five minutes is a skill worth drilling.
- **Stop early.** When only one target matters, return the moment it is finished. On random maps this often saves half the work.

6.7 Chapter Summary

- A state space turns puzzles into graphs; write down state, number of states, actions and costs before coding.
- The shape of the weights chooses the algorithm: BFS for weight 1, 0-1 BFS for weights 0 and 1, Dial's algorithm for small integers up to C , heap Dijkstra for general non-negative weights.
- Dial's algorithm is Dijkstra with a ring of $C + 1$ buckets; the window [Equation \(6.3\)](#) guarantees that no two waiting distances share a bucket.
- Lazy deletion (skip copies whose distance no longer matches) keeps both Dial and heap Dijkstra simple and fast.

- On giant grids, flat arrays and small types decide whether the program fits in time and memory.

6.8 Exercises

1. A maze contains keys `a-d` and doors `A-D`; a door can be passed only while holding its key. Every step costs 1. Describe the state, count the states for a 50×50 maze and name the algorithm.
2. Modify **Listing 6.2** so that stepping into a free square costs 1 and breaking a wall costs 3. Which algorithm from **Table 6.1** is now the natural choice, and how many buckets does it need?
3. Explain why Dial's algorithm can be wrong if some edge has weight 0 and the program uses the check "stop scanning a bucket when it becomes empty" but pushes 0-weight neighbors into the *same* bucket it is scanning. Is the program in **Listing 6.3** safe from this issue?
4. Prove that in **Listing 6.2** the deque never contains more than two distinct distances.
5. The CHIPSLL answer for a map is at most how large? Use **Equation (6.1)** and the grid size to give an upper bound and decide whether `int` is enough.

6.9 Solutions to the Exercises

1. The state must remember the square and which keys are held: $(r, c, mask)$ where $mask$ is a 4-bit set. That gives $50 \cdot 50 \cdot 2^4 = 40\,000$ states. Picking up a key changes the mask; entering a door square is allowed only if the corresponding bit is set. All steps cost 1, so plain BFS over these states is correct and fast.
2. The weights are now 1 and 3, small integers with $C = 3$, so Dial's algorithm with $C + 1 = 4$ buckets is natural. (A 0-1 BFS no longer applies, and a heap would work but is slower.)
3. With a 0-weight edge a node can be pushed into the bucket currently being scanned. A program that iterates over a *copy* of the bucket, or stops after the original contents, would miss those nodes. The CHIPSLL program pops from the bucket until it is truly empty (`while(!bkt[b].empty())`), so newly added nodes in the same bucket would be processed too; besides, CHIPSLL's weights are at least 1, so the situation never arises.
4. Induction on the number of operations. Initially the deque holds only the start, one distance. Suppose it holds distances k (front part) and possibly $k + 1$ (back part), in that order. We take a node of distance k from the front; a 0-neighbor gets distance k and goes to the front, a 1-neighbor gets $k + 1$ and goes to the back. The order and the two-value property are preserved. When the k part is used up, all nodes have distance $k + 1$, and the argument repeats with $k + 1$.
5. A route never needs to visit a square twice, and a shortest route in a grid without walls can be taken as any monotone route of $(N - 1) + (M - 1)$ steps, each costing at most 63. So the answer is at most $63 \cdot (1999 + 1999) = 251\,874$. An `int` holds values up to about $2.1 \cdot 10^9$, so it is far more than enough, and adding a weight to the "infinity" `0x3f3f3f3f` cannot overflow either.

References

- [1] Sphere Online Judge, "Problem CHIPSLL." [Online]. Available: <https://www.spoj.com/problems/CHIPSLL/>
- [2] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Lulu Press, 2020.
- [3] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

- [4] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.
- [5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 22.
- [6] R. B. Dial, "Algorithm 360: Shortest-path forest with topological ordering," *Communications of the ACM*, vol. 12, no. 11, pp. 632–633, 1969.

Chapter 7

Disjoint Sets, Kruskal's Algorithm and Sweeps — VILUNIT

Kruskal's algorithm for the minimum spanning tree is usually taught as “sort all the edges, then take them one by one”. That recipe breaks down when the graph has too many edges to write down: five thousand objects already have more than twelve million pairs. Olympiad problems love this situation, because it separates contestants who remember a recipe from those who understand why it works. This chapter builds the understanding. You will learn the disjoint-set union structure (DSU) that Kruskal's algorithm relies on, see why Kruskal's algorithm only needs edges *grouped by weight* rather than fully sorted, and use a sorted *sweep* to find the few edges that matter among millions of candidates. The case study, SPOJ VILUNIT [1], asks for the cheapest way to connect up to 5000 rectangular villages when every connection costs 0, 1 or 2, and it falls to exactly this combination.

7.1 Villages on a Map

A district office wants to link all the villages in its area into one network, for example with shared roads or a joint water system. On the map each village is a rectangle. Linking two villages costs nothing when their areas already overlap, because they are, in effect, one settlement. It costs one unit when the two rectangles lie side by side along a common strip: their shadows on the east–west axis overlap, or their shadows on the north–south axis overlap, so a straight road along that strip joins them. Any other pair costs two units. The office wants every village reachable from every other one, directly or through other villages, at the least total cost.

This is a *minimum spanning tree* problem. The villages are nodes; every pair of villages is an edge with weight 0, 1 or 2; we want a set of edges that connects everything with the smallest total weight. The obstacle is purely one of size. With $n = 5000$ villages there are $n(n - 1)/2 \approx 1.25 \cdot 10^7$ pairs per test case, and a test file contains several test cases. Computing every weight, storing every pair and sorting them would already be too slow before the real work began.

The same pattern hides in many practical tasks: merging overlapping bookings into blocks, grouping fields whose irrigation channels cross, or clustering shop locations that share a street. Whenever the “distance” between two objects takes only a few values, and those values come from geometry, there is usually a way to avoid looking at all pairs.

The plan of the chapter follows the plan of the solution. Section 7.2 builds the three tools from zero: the disjoint-set union, Kruskal's algorithm seen as a sequence of weight classes, and the interval sweep. Section 7.3 runs Kruskal's algorithm on a small graph so that its decisions can be checked by hand. Section 7.4 combines the tools on VILUNIT, with a full trace of a sample, the complete program and an honest look at its running time. Section 7.5 tests it against the definition written out as a brute force, and Section 7.6 shows where else the same reasoning applies. Each tool is short; the lesson is in how they fit together.

7.2 The Tools from Zero

7.2.1 Disjoint-set union in five lines

A *disjoint-set union* (DSU, also called union–find) maintains a partition of the objects $0, 1, \dots, n - 1$ into groups. It supports two operations: `find(x)` returns a representative of the group containing x , and `unite(a, b)` merges the groups of a and b . The representation is a forest: every object stores a parent `par[x]`, and a root (an object with `par[x] == x`) represents its tree.

DEFINITION. *Path compression* makes every node visited by `find` point directly to the root. *Union by rank* (or by size) always attaches the smaller tree under the larger one. With both, any sequence of m operations on n objects takes $O(m\alpha(n))$ time, where α is the inverse Ackermann function, which is at most 4 for every input that fits in the universe [2], [3].

In contest code path compression alone is common: its worst case is $O(\log n)$ amortized per operation, and in practice it is indistinguishable from the full version. The five lines used throughout this chapter are

```
int find(int x){ return par[x]==x ? x : par[x]=find(par[x]); }
bool unite(int a,int b){ a=find(a); b=find(b);
    if(a!=b){ par[a]=b; return true; } return false; }
```

The Boolean result of `unite` is the key to Kruskal's algorithm: `true` means two different groups were merged (the edge is useful), `false` means the two objects were already connected (the edge would close a cycle). Counting groups at the end is equally simple: count the objects i with `par[i] == i`.

7.2.2 Kruskal's algorithm and the cut property

Kruskal's algorithm [4] considers the edges in order of increasing weight and keeps an edge exactly when it joins two different groups. Why does this greedy choice give a minimum spanning tree? The standard argument uses the *cut property*: for any division of the nodes into two sides, a cheapest edge crossing between the sides belongs to some minimum spanning tree [3]. When Kruskal's algorithm accepts an edge (u, v) , take one side to be u 's current group. No cheaper edge crosses this cut, since any cheaper crossing edge would have been examined earlier and accepted, merging the groups. So the edge is safe, and by induction every accepted edge keeps the chosen set inside some minimum spanning tree.

Now look at what the argument really needs. It needs cheaper edges to come before more expensive ones. It does *not* care in which order edges of the *same* weight are examined. Moreover, the number of edges accepted within one weight class is fixed: if there are k_{w-1} groups after all edges lighter than w and k_w groups after the edges of weight w as well, then exactly $k_{w-1} - k_w$ edges of weight w were accepted, whichever ones they were. The total weight is therefore

$$\text{MST} = \sum_w w \cdot (k_{w-1} - k_w) \quad (7.1)$$

where k_w is the number of connected components of the graph that keeps only the edges of weight at most w . Equation (7.1) turns the minimum spanning tree into a question about *connectivity*, one weight class at a time. For weights 0, 1 and 2 and a complete graph (every pair has some edge) it reads

$$\text{MST} = 1 \cdot (k_0 - k_1) + 2 \cdot (k_1 - 1) \quad (7.2)$$

because in the end everything is connected, so $k_2 = 1$. The weight-0 class contributes nothing, but it still matters: it decides k_0 .

7.2.3 Sweeping intervals

So we need connectivity, not a list of edges. For intervals on a line this is easy. Call two intervals *overlapping* when they share a piece of positive length. Sort the intervals by left end and walk through them, remembering `cur_max`, the farthest right end seen so far. The next interval $[l, r]$ overlaps some earlier interval exactly when $l < \text{cur_max}$; and if it does, it overlaps the interval that achieved `cur_max`, because that interval started no later than l and ends after l . If $l \geq \text{cur_max}$, then every earlier interval ended at or before l , and a new group starts.

KEY INSIGHT: Overlap spreads through chains. If A overlaps B and B overlaps C , the three are in one group even when A and C are far apart. Uniting each interval with the *representative* that holds `cur_max` builds exactly these groups with $n - 1$ unions at most, instead of checking n^2 pairs.

Listing 7.1 shows the sweep on seven intervals.

Listing 7.1 `interval_groups.cpp`: grouping intervals by a sweep with the farthest right end.

```
// Sweep: group intervals that overlap directly or through a chain.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<pair<int, int>> iv = {{1, 4}, {9, 12}, {3, 6}, {14, 15},
                                {5, 8}, {11, 13}, {16, 18}};
    sort(iv.begin(), iv.end()); // by left end
    int groups = 0, cur_max = -1000000000;
    for (auto [l, r] : iv) {
        if (l >= cur_max) groups++; // starts after everything seen: new group
        cur_max = max(cur_max, r); // farthest right end so far
        cout << "[" << l << ", " << r << "]" << " -> group " << groups << "\n";
    }
    cout << groups << " groups\n";
}
```

```
[1,4] -> group 1
[3,6] -> group 1
[5,8] -> group 1
[9,12] -> group 2
[11,13] -> group 2
[14,15] -> group 3
[16,18] -> group 4
4 groups
```

$[1,4]$ and $[5,8]$ do not overlap, yet they share group 1 through $[3,6]$. It is worth asking why the program remembers the interval with the *farthest right end* rather than simply the previous interval. Consider $[0,10]$, $[1,2]$ and $[5,6]$ in that order. The third interval does not overlap the second, which ended at 2, but it does overlap the first, which reaches 10. A sweep that compared each interval only with its predecessor would start a new group at $[5,6]$ and be wrong. The farthest right end summarizes everything that came before in one number: if the new interval starts before it, there is an overlap with the interval that owns it; if not, there is no overlap with anything seen so far. That single number is what makes the sweep linear after sorting. The condition is strict: $[14,15]$ and $[16,18]$ would stay apart even if the second started at 15, because touching at a single point is not an overlap of positive length.

7.2.4 Why the DSU is fast enough

It is worth seeing why path compression alone is fast, because the argument tells you when it is *not*. Without compression, `find` walks from a node to its root, and a careless sequence of unions can build a single chain of length n ; then every `find` costs n steps and n operations cost n^2 . With compression, the first `find` on that chain costs n steps but leaves every visited node pointing directly at the root, so the next `find` on any of them costs one step. Expensive operations pay for making later ones cheap. The careful accounting of this idea gives an amortized bound of $O(\log n)$ per operation for compression alone, and the famous $O(\alpha(n))$ bound once union by rank is added [2].

For contest sizes the difference is invisible: with $n = 5000$, even a pessimistic $\log_2 n \approx 13$ steps per operation is nothing. The one place where compression alone can hurt is deep *recursion*: the recursive `find` of Section 7.2.1 recurses once per node on the path, and a chain of 10^6 nodes built by unlucky unions can overflow the stack on its first compression. When n is that large, add union by size (a few more lines) or write `find` with a loop. For VILUNIT, with $n \leq 5000$, the five lines are perfect.

A second property is used silently throughout this chapter: the DSU never *splits* groups. Once two objects are connected they stay connected. That is exactly the behavior Kruskal's algorithm needs, and it is also why a DSU cannot answer questions about deleting edges; problems that delete edges are usually solved by processing the operations backward, turning deletions into insertions.

7.3 Warm-Up: Kruskal on a Small Graph

Before combining the pieces, run the classic algorithm once. **Listing 7.2** sorts nine edges of a six-node graph and reports every decision.

Listing 7.2 `kruskal.cpp`: Kruskal's algorithm with the five-line DSU.

```
// Kruskal with a five-line DSU on a small weighted graph.
#include <algorithm>
#include <iostream>
#include <tuple>
#include <vector>
using namespace std;
int par[6];
int find(int x) { return par[x] == x ? x : par[x] = find(par[x]); }
bool unite(int a, int b) {
    a = find(a), b = find(b);
    if (a == b) return false; // already connected: edge would close a cycle
    par[a] = b;
    return true;
}
int main() {
    vector<tuple<int, int, int>> e = {{4, 0, 1}, {1, 0, 2}, {3, 1, 2}, {2, 1, 3},
                                     {5, 2, 3}, {7, 3, 4}, {6, 2, 4}, {2, 4, 5},
                                     {8, 3, 5}};

    sort(e.begin(), e.end());
    for (int i = 0; i < 6; i++) par[i] = i;
    int total = 0;
    for (auto [w, u, v] : e) {
        bool used = unite(u, v);
        cout << "edge " << u << "-" << v << " w=" << w
              << (used ? " take" : " skip (cycle)") << "\n";
        if (used) total += w;
    }
    cout << "MST weight = " << total << "\n";
}
```

```
edge 0-2 w=1 take
edge 1-3 w=2 take
edge 4-5 w=2 take
edge 1-2 w=3 take
edge 0-1 w=4 skip (cycle)
edge 2-3 w=5 skip (cycle)
edge 2-4 w=6 take
edge 3-4 w=7 skip (cycle)
edge 3-5 w=8 skip (cycle)
MST weight = 14
```

Five edges are taken for six nodes, as a spanning tree requires. Check [Equation \(7.1\)](#) against the output: after the weight-1 edge there are 5 groups, after the weight-2 edges 3, after weight 3 two, and after weight 6 one. The sum $1 \cdot (6 - 5) + 2 \cdot (5 - 3) + 3 \cdot (3 - 2) + 6 \cdot (2 - 1) = 1 + 4 + 3 + 6 = 14$ matches.

7.4 Case Study: VILUNIT

7.4.1 The problem in our own words

The problem is SPOJ VILUNIT [1]. The input begins with the number of test cases. Each test case gives n ($n \leq 5000$) and then n villages, each as four integers x_1, y_1, x_2, y_2 : two opposite corners of an axis-parallel rectangle, in either order. For two villages, the cost of linking them is

- 0 if their x-projections overlap *and* their y-projections overlap (their interiors overlap);
- 1 if exactly one of the two projections overlaps;
- 2 otherwise.

Here “overlap” means sharing a segment of positive length, as in Section 7.2.3. The answer for a test case is the minimum total cost of a set of links that connects all villages, which is the weight of a minimum spanning tree. For $n \leq 1$ the answer is 0.

Figure 7.1 shows five villages and their shadows on both axes. A and B overlap, so linking them is free. C shares a y -strip with A , D shares an x -strip with A and B , so each of those links costs 1. E shares no strip with anyone and needs a link of cost 2. By Equation (7.2), $k_0 = 4$ ($\{A, B\}, C, D, E$), $k_1 = 2$, and the answer is $1 \cdot 2 + 2 \cdot 1 = 4$.

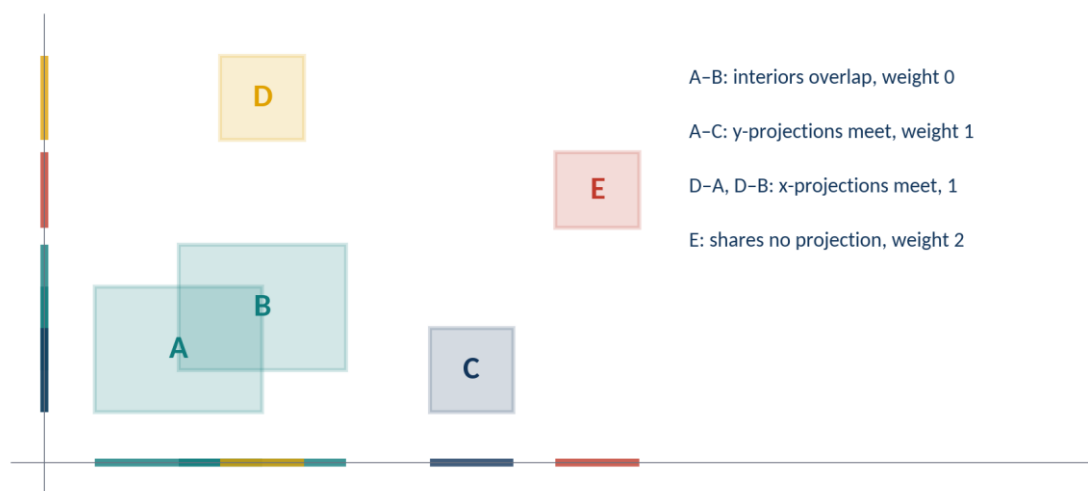


Figure 7.1 Five villages and their projections (colored bars on the axes); the link costs 0, 1 and 2 come from which projections overlap.

7.4.2 Sample input and output

```
2
3
0 0 4 4
2 2 6 6
10 0 12 2
4
0 0 2 2
5 5 7 7
1 5 3 9
9 1 11 3
```

```
1
3
```

In the first test the first two squares overlap (cost 0) and the third shares the y -strip from 0 to 2 with the first (cost 1): total 1. In the second test no interiors overlap. The third village shares an x -strip with the first ($[1,3]$ against $[0,2]$) and a y -strip with the second ($[5,9]$ against $[5,7]$); the fourth shares a y -strip with the first ($[1,3]$ against $[0,2]$). Three links of cost 1 connect all four: total 3.

7.4.3 Designing the solution

Equation (7.2) says we need only two numbers: k_0 , the number of groups formed by free links, and k_1 , the number formed by links of cost at most 1.

1. **Normalize** each rectangle so that $x_1 \leq x_2$ and $y_1 \leq y_2$.
2. **Pass 0 (weight 0)**. Sort by x_1 . For each village u , scan the following villages v in this order while $x_1(v) < x_2(u)$; those are exactly the villages whose x -projection overlaps u 's from the right. If their y -projections overlap too, unite them.
3. **Weight 1, the x -sweep**. Walk the villages in order of x_1 with `cur_max` and its owner `rep` as in Section 7.2.3; when $x_1(u) < cur_max$, unite u with `rep`, and if the union succeeds add 1 to the cost.

4. Weight 1, the y-sweep. The same with y_1, y_2 and a second sorted order.

5. Weight 2. Count the remaining groups k_1 and add $2(k_1 - 1)$.

Each successful union in step 3 or 4 joins two different groups through a real link of cost 1: the link exists because the projections overlap, and its cost is not 0, since two villages with overlapping interiors were already united in step 2. The number of successful unions is therefore $k_0 - k_1$, as Equation (7.2) requires, *provided* the two sweeps together discover the full connectivity of the weight- ≤ 1 graph. They do: the x-sweep unites every group of the x-overlap graph, the y-sweep every group of the y-overlap graph, and the DSU combines both, which is exactly connectivity in the union of the two graphs.

A fair question is whether all this cleverness is needed. For a *dense* graph, where every pair of nodes has an edge, the textbook alternative is Prim's algorithm in its array form: keep for every node outside the tree its cheapest link into the tree, add the node with the cheapest link, and update the others in one pass. That costs $O(n^2)$ time and $O(n)$ memory, and it never stores an edge list, because each weight is computed from the coordinates when it is needed [3]. For $n = 5000$ this is $2.5 \cdot 10^7$ weight evaluations per test case, which is feasible for a few test cases but becomes risky when a file holds many large ones. The weight-class method is faster in the common case and, more importantly, its idea scales: with $n = 10^5$ villages the array form of Prim's algorithm would need 10^{10} steps, while the sweeps would still run in a fraction of a second if pass 0 were replaced by a sweep-line structure. In a contest, knowing both is the real strength: Prim's array form is a five-minute brute force that can even serve as the stress-test partner, and the sweeps are the solution that survives the largest tests.

7.4.4 Tracing the second sample by hand

Before reading the code, trace the design on the second sample test, whose four villages, after normalization, are $V_0 = [0,2] \times [0,2]$, $V_1 = [5,7] \times [5,7]$, $V_2 = [1,3] \times [5,9]$ and $V_3 = [9,11] \times [1,3]$.

Pass 0. In order of x_1 the villages are V_0 (0), V_2 (1), V_1 (5), V_3 (9). For V_0 the scan looks at V_2 , since $1 < 2$; their y-projections $[0,2]$ and $[5,9]$ do not overlap, so nothing happens, and the scan stops at V_1 because $5 \geq 2$. The other villages find no candidates. No interiors overlap, so $k_0 = 4$.

The x-sweep. V_0 sets `cur_max` to 2 with representative V_0 . V_2 starts at $1 < 2$: unite V_2 with V_0 , cost 1; `cur_max` becomes 3 with representative V_2 . V_1 starts at $5 \geq 3$: a new group; `cur_max` becomes 7. V_3 starts at $9 \geq 7$: a new group again.

The y-sweep. In order of y_1 : V_0 (0), V_3 (1), then V_1 and V_2 (both 5). V_0 sets `cur_max` to 2. V_3 starts at $1 < 2$: unite with V_0 , cost 2; `cur_max` becomes 3. The next village starts at $5 \geq 3$ and only raises `cur_max`; the last one starts at 5, below the new `cur_max`, and joins it: cost 3. Whichever of V_1, V_2 comes first after sorting, the outcome is the same.

The second round of the `changed` loop finds nothing new, one group remains, and the answer is $3 + 2 \cdot (1 - 1) = 3$, as expected. Tracing a sample this way takes five minutes and is the best protection against a sweep that runs over the wrong index array.

7.4.5 The complete solution

Listing 7.3 is the complete program, recorded as *Accepted* on SPOJ. The P50 block contains the two weight-1 sweeps.

Listing 7.3 `VILUNIT.cpp`: Kruskal by weight classes, with pass 0 and two sweeps.

```
#include <stdio>
#include <algorithm>
#include <vector>
using namespace std;
#define MAXN 5005
#define IN_BUF (1 << 20)
static int X1[MAXN], Y1[MAXN], X2[MAXN], Y2[MAXN];
static int par[MAXN], idxX[MAXN], idxY[MAXN];
static char ibuf[IN_BUF];
static int ipos, ilen;
static inline int readInt() {
    int x = 0, f = 1;
    while (ipos < ilen && (ibuf[ipos] < '0' || ibuf[ipos] > '9')) {
        if (ibuf[ipos] == '-') f = -1;
        if (++ipos == ilen) {
            ilen = fread(ibuf, 1, IN_BUF, stdin);
            ipos = 0;
            if (ilen <= 0) return 0;
        }
    }
    while (ipos < ilen && ibuf[ipos] >= '0' && ibuf[ipos] <= '9') {
        x = x * 10 + (ibuf[ipos++] - '0');
        if (ipos == ilen) {
            ilen = fread(ibuf, 1, IN_BUF, stdin);
            ipos = 0;
        }
    }
    return x * f;
}
int find(int x) {
    return par[x] == x ? x : par[x] = find(par[x]);
}
bool unite(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b) {
        par[a] = b;
        return true;
    }
    return false;
}
int main() {
    ilen = fread(ibuf, 1, IN_BUF, stdin);
    int T = readInt();
    while (T--) {
        int n = readInt();
```

Listing 7.3 (Continued) *VILUNIT.cpp: Kruskal by weight classes, with pass 0 and two sweeps.*

```

if (n <= 1) {
    if (n == 1) {
        readInt();
        readInt();
        readInt();
        readInt();
    }
    printf("0\n");
    continue;
}
for (int i = 0; i < n; i++) {
    X1[i] = readInt();
    Y1[i] = readInt();
    X2[i] = readInt();
    Y2[i] = readInt();
    if (X1[i] > X2[i]) swap(X1[i], X2[i]);
    if (Y1[i] > Y2[i]) swap(Y1[i], Y2[i]);
    par[i] = i;
    idxX[i] = i;
    idxY[i] = i;
}
sort(idxX, idxX + n, [](int a, int b) {
    return X1[a] < X1[b];
});
sort(idxY, idxY + n, [](int a, int b) {
    return Y1[a] < Y1[b];
});
// Pass 0: Weight 0 (Overlapping Interiors)
for (int i = 0; i < n; i++) {
    int u = idxX[i];
    for (int j = i + 1; j < n; j++) {
        int v = idxX[j];
        if (X1[v] >= X2[u]) break;
        if (Y1[v] < Y2[u] && Y2[v] > Y1[u]) unite(u, v);
    }
}
long long cost = 0;
bool changed = true;
while (changed) {
    changed = false;
    // <P50>
    // Weight-1 sweeps: walk idxX (then idxY) keeping the farthest right
    // end cur_max and its owner rep; a box starting before cur_max joins
    // rep, and each successful unite adds 1 to cost and sets changed.
    // X-Sweep (Weight 1)
    // >>> YOUR TURN (Public Edition): write this part yourself, following the hint
    // above. <<<

```

Listing 7.3 (Continued) *VILUNIT.cpp: Kruskal by weight classes, with pass 0 and two sweeps.*

```

// </P50>
}
int comps = 0;
for (int i = 0; i < n; i++) if (par[i] == i) comps++;
printf("%lld\n", cost + 2LL * (comps - 1));
}
return 0;
}

```

7.4.6 Line by line

Fast input. The input can be large: many test cases with up to 5000 lines each. `readInt` reads the whole input in blocks of one megabyte with `fread` and parses digits by hand, which is several

times faster than `scanf`. It skips any non-digit characters, remembers a minus sign, and refills the buffer whenever it runs out.

The DSU. `find` and `unite` are the five lines of Section 7.2.1, working on the global `par` array.

Small cases. When $n \leq 1$ the program consumes the one village (if any) and prints 0 at once. Without this, $n = 0$ would reach the closing formula with zero groups and print -2 .

Reading and normalizing. Each rectangle is read, its corners are swapped where needed, `par[i] = i` makes every village its own group, and two index arrays are prepared for sorting. Sorting *indices* instead of the rectangles themselves keeps the four coordinate arrays untouched, so the x-order and the y-order can coexist.

Pass 0. The double loop implements step 2 of the design. The `break` when $x_1(v) \geq x_2(u)$ is correct because the order is sorted by x_1 : every later village starts even further right. The y-test `Y1[v] < Y2[u] && Y2[v] > Y1[u]` is the standard overlap test for two intervals.

The sweeps (the P50 block). The `while (changed)` loop repeats the x-sweep and the y-sweep until neither performs a successful union. Inside, each sweep is the interval sweep of **Listing 7.1**, with `unite(u, rep)` in place of the group counter and `cost++` for every union that succeeds. The starting value $-2 \cdot 10^9$ lies below every coordinate.

The answer. The last lines count the roots, which is k_1 , and print `cost + 2LL * (comps - 1)`, which is **Equation (7.2)**.

7.4.7 Complexity, honestly

The two sorts cost $O(n \log n)$ and each sweep is $O(n \alpha(n))$. Pass 0 is different: its inner loop runs over all villages that overlap u along the x-axis, and when all villages are long and thin in the x-direction that is every village, so the worst case is $O(n^2)$, about $1.25 \cdot 10^7$ cheap comparisons for $n = 5000$. We measured the worst kind of input, twenty test cases of 5000 huge random rectangles that all overlap, and the program finished in about 1.2 seconds, so the quadratic pass is affordable at this size. The `changed` loop runs its body at most twice in practice; Exercise 2 shows that one round is always enough.

7.4.8 Traps

PITFALL: Counting every union as cost 1 *before* pass 0 has run. If the weight-1 sweep runs first, it may join two villages whose interiors overlap and charge 1 for a free link. Kruskal's order, cheapest class first, is not optional.

- **Touching is not overlapping.** Every comparison is strict (`<`, `>`). Using `<=` changes the meaning and the answers.
- **Corners in any order.** Without the swaps a rectangle given from top-right to bottom-left becomes an "interval" with negative length and silently breaks both passes.
- **Forgetting $n = 0$.** The formula $2(k_1 - 1)$ assumes at least one group.
- **Overflow.** The cost can reach $2(n - 1)$, small here, but the program still uses `long long` for the sum; a habit worth keeping.

7.5 Stress Testing Against a Brute Force

The brute force in **Listing 7.4** is the definition written out: compute all $n(n-1)/2$ weights directly from the projections, sort, and run Kruskal. It shares nothing with the sweeps, which makes it a trustworthy judge.

Listing 7.4 `brute.cpp`: all pairs, explicit weights 0, 1, 2, and Kruskal.

```
// Reference for VILUNIT: all n(n-1)/2 pairs with weights 0/1/2, then Kruskal.
#include <algorithm>
#include <cstdio>
#include <numeric>
#include <tuple>
#include <vector>
using namespace std;
vector<int> par;
int find(int x) { return par[x] == x ? x : par[x] = find(par[x]); }
int main() {
    int T;
    if (scanf("%d", &T) != 1) return 0;
    while (T--) {
        int n;
        if (scanf("%d", &n) != 1) return 0;
        vector<int> a(n), b(n), c(n), d(n);
        for (int i = 0; i < n; i++) {
            if (scanf("%d%d%d%d", &a[i], &b[i], &c[i], &d[i]) != 4) return 0;
            if (a[i] > c[i]) swap(a[i], c[i]);
            if (b[i] > d[i]) swap(b[i], d[i]);
        }
        vector<tuple<int, int, int>> e;
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j < n; j++) {
                bool ox = a[i] < c[j] && a[j] < c[i]; // x-projections share length
                bool oy = b[i] < d[j] && b[j] < d[i]; // y-projections share length
                e.emplace_back(ox && oy ? 0 : (ox || oy ? 1 : 2), i, j);
            }
        sort(e.begin(), e.end());
        par.assign(n, 0);
        iota(par.begin(), par.end(), 0);
        long long cost = 0;
        for (auto [w, i, j] : e)
            if (find(i) != find(j)) par[find(i)] = find(j), cost += w;
        printf("%lld\n", cost);
    }
}
```

The generator **Listing 7.5** draws up to nine rectangles with positive width and height in a small square, so that overlaps, shared strips and touching edges all happen often. The coordinate range is chosen at random from 6, 12 and 30 to vary the density.

Listing 7.5 `gen.py`: random small villages for VILUNIT.

```
# Random small VILUNIT inputs (boxes with positive area): python3 gen.py SEED
import random, sys
random.seed(int(sys.argv[1]))
T = random.randint(1, 5)
print(T)
for _ in range(T):
    n = random.randint(1, 9)
    L = random.choice([6, 12, 30])
    print(n)
    for _ in range(n):
        x1, x2 = random.sample(range(L), 2)
        y1, y2 = random.sample(range(L), 2)
        print(x1, y1, x2, y2)
```

Running the same `stress.sh` as in Chapter 6 with the problem name VILUNIT gives

```
all 300 random tests passed
```

7.6 Using the Idea Beyond VILUNIT

The combination “few weight classes, connectivity per class” appears in many disguises. Three are worth knowing.

Thresholds instead of classes. “What is the smallest L such that all objects are connected using only links of cost at most L ?” is answered by Kruskal’s algorithm too: L is the weight of the last edge accepted, the *bottleneck* of the minimum spanning tree. If the candidate thresholds are few, run the connectivity test once per threshold, as the sweeps did for threshold 1.

Nearest neighbors on a line. When objects are points on a line and the cost of a link is their distance, only neighboring points in sorted order can be useful: any link that jumps over a point can be replaced by two shorter links through that point. So a line of n points has only $n - 1$ candidate edges, and the minimum spanning tree is simply the sum of the gaps. The same “sort first, then only neighbors matter” reasoning is what made the sweeps work.

Manhattan distance in the plane. For points with cost $|x_1 - x_2| + |y_1 - y_2|$, a classic theorem shows that each point needs to be linked only to its nearest neighbor in each of eight angular regions, giving $O(n)$ candidate edges that a sweep with a Fenwick tree finds in $O(n \log n)$ [5]. The details are longer, but the philosophy is the one of this chapter: never list all pairs; let geometry pick the edges that matter, and let the DSU do the rest.

In each case, a sentence from Section 7.2.2 is the starting point: Kruskal’s algorithm needs only the *order* of weights and the *connectivity* inside each class. Anything that provides these two things, without a list of all pairs, is a valid implementation.

7.7 Contest-Day Tactics

- **Few distinct weights? Think in classes.** When an MST problem has weights from a tiny set, write Equation (7.1) on paper and ask how to compute the number of components per class, not how to list edges.
- **Look for geometry that sorts.** Overlap of intervals, reachability on a line and “next greater element” are all solved by one sorted pass. Two axes usually mean two passes.
- **Keep the DSU in muscle memory.** It is the most reused structure in olympiad graph problems [5]; typing it must take less than a minute.

- **Test degenerate inputs.** $n = 0$, $n = 1$, all villages identical, all villages in one row. These are where formulas with “ -1 ” break.
- **Measure the worst case you suspect.** If a loop is quadratic in theory, build that input and time it, as in Section 7.4.7. Knowing the real cost beats hoping.
- **Keep a dense brute force at hand.** Prim's algorithm in array form, with weights computed on the fly, is a trustworthy second opinion for any minimum spanning tree on up to a few thousand objects, and it needs no edge list at all.

7.8 Chapter Summary

- The DSU with path compression answers “are these connected?” and “join these” in almost constant time; its Boolean `unite` is Kruskal's decision.
- Kruskal's algorithm only needs weight classes in increasing order; within a class, any order gives the same number of accepted edges, which yields Equation (7.1).
- Sorted sweeps with a farthest-right-end representative find interval connectivity with at most $n - 1$ unions.
- VILUNIT combines all three: free links in pass 0, cost-1 links in two sweeps, and the rest by arithmetic, as in Equation (7.2).

7.9 Exercises

1. Run the interval sweep of Listing 7.1 by hand on [2,5], [1,3], [6,7], [4,6]. How many groups are there?
2. Prove or refute: in Listing 7.3 a single round of x-sweep followed by y-sweep is always enough, so the `changed` loop can be removed.
3. Suppose linking costs were 0, 1 and 5 instead of 0, 1 and 2. Which single line of the program changes?
4. Remove the two `swap` lines from Listing 7.3. Find the correct answer for the villages 4 2 2 1, 7 7 3 6 and 0 3 7 6, and explain why the modified program gets it wrong.
5. Add union by size to the DSU of Listing 7.2. What changes in the output?
6. A test case has $n = 5000$ villages, no two of which share any projection. What does the program print, and how much work does pass 0 do?

7.10 Solutions to the Exercises

1. Sorted: [1,3], [2,5], [4,6], [6,7]. [1,3] starts group 1, `cur_max` becomes 3. [2,5]: $2 < 3$, same group, `cur_max` 5. [4,6]: $4 < 5$, same group, `cur_max` 6. [6,7]: $6 < 6$ is false, new group. Two groups.
2. True. After the x-sweep, every pair of villages with overlapping x-projections lies in one group, because the sweep builds the connected groups of the x-overlap graph (Section 7.2.3). After the y-sweep the same holds for the y-overlap graph. Groups only merge, never split, so after both sweeps every edge of the union of the two graphs joins two objects of the same group, which means the DSU groups are exactly the components of the weight- ≤ 1 graph. A second round can find nothing new; it serves only as a safety net.
3. Only the last print statement: `cost + 5LL * (comps - 1)`. The classes are still processed in increasing order, and Equation (7.1) with $w = 5$ gives that term.

4. Normalized, the villages are $A = [2,4] \times [1,2]$, $B = [3,7] \times [6,7]$ and $C = [0,7] \times [3,6]$. No interiors overlap. A and C share an x-strip, B and C share an x-strip, and B and C only *touch* along $y = 6$. Two links of cost 1 connect everything, so the answer is 2, which the full program prints. Without the swaps, B is stored with $x_1 = 7$ and $x_2 = 3$, so the x-sweep believes B starts at 7. At that moment `cur_max` is 7 (the right end of C), the strict test $7 < 7$ fails, and B is not joined to C . The y-sweep cannot help, because B and C only touch. B stays alone, the program pays 2 for it instead of 1, and prints 3.
5. Nothing in the output: union by size changes the shape of the DSU trees, not which edges join different groups, so the same edges are taken and the weight stays 14. Only the running time on large inputs improves.
6. Every pair costs 2, so the answer is $2(n - 1) = 9998$. Pass 0 is cheap here: for each village the scan stops at the first following village in x-order, because no x-projections overlap, so the double loop does about n comparisons. The quadratic worst case needs many overlapping x-projections, the opposite situation.

References

- [1] Sphere Online Judge, "Problem VILUNIT." [Online]. Available: <https://www.spoj.com/problems/VILUNIT/>
- [2] 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.
- [3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 19 and 21.
- [4] 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.
- [5] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Lulu Press, 2020.

Chapter 8

Euler Tours, Fenwick Trees and Binary Lifting — HBD

Some olympiad problems are not solved by one clever idea but by three ordinary ones that fit together like gears. This chapter teaches that way of working. You will learn to flatten a rooted tree into an array with an *Euler tour*, so that every subtree becomes a contiguous interval; to keep sums over those intervals with a *Fenwick tree* (binary indexed tree), so that the sum of values on the path from the root to any node is one prefix query; and to climb a tree in steps of 2^j with *binary lifting*, so that a search along a path takes logarithmic time. Each gear is short and easy on its own. The skill that wins medals is seeing that a problem is asking for all three at once and writing them down as clean contracts. The case study, SPOJ HBD [1], asks exactly this: on a tree with up to 10^5 nodes whose values change, how many steps upward from a node are needed before the collected values reach a target?

8.1 Collecting Donations Up the Family Tree

A large family is organizing a celebration. The family tree is drawn with the eldest ancestor at the top. Each person has promised some amount of money, and promises grow as the date comes closer: an uncle adds fifty thousand rupiah, a cousin adds twenty. The organizer often asks questions of one shape: “Starting from this grandchild and going up through the parents, grandparents and so on, but no higher than this particular ancestor, how many people must contribute before we have collected at least this amount?”

With a handful of people, the organizer walks up the tree and adds as she goes. With a hundred thousand people and a hundred thousand mixed updates and questions, walking is too slow: a family tree can be a long chain, and each question might walk tens of thousands of steps. We need a way to know the total along any upward path instantly, even while the values change, and a way to find the stopping point without walking. The same need appears in company hierarchies (how many levels of managers must approve a budget before the approved total reaches a threshold), in file systems (folder sizes along a path) and in network routing trees.

The chapter is organized around the three gears. Section 8.2 builds each of them from zero and proves the identity that connects it to the next: subtrees are intervals, root-to-node sums are prefix sums, vertical path sums are differences, and monotone searches along a path are sequences of power-of-two jumps. Section 8.3 runs all three on a nine-node tree whose numbers can be checked by hand. Section 8.4 composes them into a solution of HBD, traces one question step by step and explains every line of the program. Section 8.5 tests it against the organizer’s slow walk, and Section 8.6 lists the other problems that the same gears solve.

8.2 The Three Gears from Zero

Throughout, the tree is *rooted* at node 1. The *parent* of a node is its neighbor one step closer to the root, the *ancestors* of a node are its parent, its parent’s parent and so on up to the root (we also

count the node itself as its own ancestor when convenient), and the *subtree* of u is u together with everything below it. The *depth* of the root is 1, and each step down adds 1.

8.2.1 Gear 1: the Euler tour turns subtrees into intervals

Run a depth-first search from the root and keep a clock. When the search first enters a node u , advance the clock and write the time as $tin[u]$. When the search leaves u for the last time, write the current clock value as $tout[u]$, without advancing it. Every node gets a distinct entry time between 1 and n . Follow the clock on the tree of **Figure 8.1**, visiting children in increasing order. The search enters 1 (time 1), then 2 (time 2), then 4 (time 3); node 4 has no children, so it is left with $tout[4] = 3$. Back at 2, the search enters 5 (time 4), then 7 (time 5), then 9 (time 6), which is a leaf: $tout[9] = 6$ and, since 7 has no other child, $tout[7] = 6$ as well. Back at 5 it enters 8 (time 7), a leaf, and then leaves 8, 5 and 2 with exit time 7. Finally it enters 3 (time 8) and 6 (time 9) and leaves 6, 3 and 1 with exit time 9. The exit time of a node is always the entry time of the last node of its subtree.

Because depth-first search finishes the whole subtree of u before it leaves u , the nodes of that subtree are exactly the nodes entered between u 's entry and its exit:

$$w \text{ is in the subtree of } u \Leftrightarrow tin[u] \leq tin[w] \leq tout[u] \quad (8.1)$$

Figure 8.1 shows a nine-node tree with every node labeled $[tin, tout]$, and below it the nodes written in order of entry time. The subtree of node 5, namely $\{5, 7, 9, 8\}$, occupies positions 4 to 7: one interval.

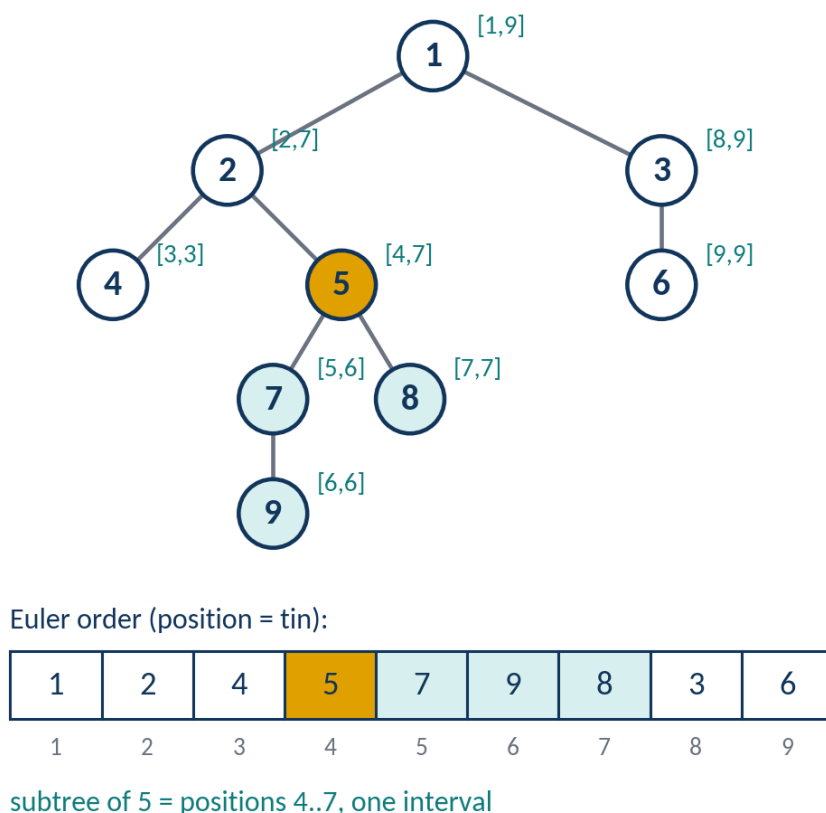


Figure 8.1 A tree with entry and exit times $[tin, tout]$; in entry order, the subtree of node 5 fills the single interval of positions 4 to 7.

8.2.2 Gear 2: a Fenwick tree for prefix sums with updates

A *Fenwick tree* [2] stores an array $a[1..n]$ so that both “add v to $a[i]$ ” and “compute $a[1] + \dots + a[i]$ ” take $O(\log n)$ time. Its storage is an array `bit` where `bit[i]` holds the sum of a over the block of length $\text{low}(i)$ ending at i , and $\text{low}(i)$ is the value of the lowest set bit of i , computed in C++ as `i & -i`. A prefix sum walks down by removing the lowest bit each time; an update walks up by adding it:

```
add(i, v): while i <= n: bit[i] += v; i += i & -i
query(i):  s = 0; while i > 0: s += bit[i]; i -= i & -i; return s
```

Each loop runs at most $\lceil \log_2 n \rceil + 1$ times, because every step changes the lowest set bit.

The trick that matters here is to use the Fenwick tree in reverse: *range update, point query*. Give each node u the value $\text{val}[u]$ and perform

$$\text{add}(\text{tin}[u], +\text{val}[u]), \quad \text{add}(\text{tout}[u] + 1, -\text{val}[u]) \quad (8.2)$$

The prefix sum up to position $\text{tin}[w]$ then collects $+\text{val}[u]$ exactly when $\text{tin}[u] \leq \text{tin}[w]$ and does not collect the matching $-\text{val}[u]$ exactly when $\text{tin}[w] \leq \text{tout}[u]$. By Equation (8.1) that happens precisely when w lies in the subtree of u , that is, when u is an ancestor of w . So

$$S(w) = \text{query}(\text{tin}[w]) = \sum_{u \text{ ancestor of } w} \text{val}[u] \quad (8.3)$$

is the sum of values on the path from the root down to w . When a value changes by y , applying Equation (8.2) with y updates every affected $S(w)$ at once, in $O(\log n)$.

8.2.3 Path sums as differences

Let a be an ancestor of v . The nodes on the upward path from v to a , both included, are the ancestors of v minus the ancestors of $\text{parent}(a)$. Hence

$$\text{sum}(v \rightarrow a) = S(v) - S(\text{parent}(a)) \quad (8.4)$$

with the convention $S(0) = 0$ for the imaginary parent of the root. Every question about a vertical path is now a subtraction of two prefix sums. For example, on the tree of **Figure 8.1** with the values of Section 8.3, the path from node 9 up to node 5 sums to $S(9) - S(2) = 23 - 8 = 15$, that is $8 + 6 + 1$, without walking a single step. This is the contract of the first two gears: *give me any node, and I tell you the root-to-node sum in $O(\log n)$, even while values change*.

8.2.4 A Fenwick tree by hand

The Fenwick tree looks like magic until you draw its blocks once. For $n = 9$, the value of `i & -i` is 1 for odd i , 2 for $i = 2, 6$, 4 for $i = 4$ and 8 for $i = 8$. So `bit[1]` covers $a[1]$, `bit[2]` covers $a[1..2]$, `bit[3]` covers $a[3]$, `bit[4]` covers $a[1..4]$, `bit[5]` covers $a[5]$, `bit[6]` covers $a[5..6]$, `bit[7]` covers $a[7]$, `bit[8]` covers $a[1..8]$ and `bit[9]` covers $a[9]$.

A prefix query for position 7 walks $7 \rightarrow 6 \rightarrow 4 \rightarrow 0$: it adds `bit[7]`, `bit[6]` and `bit[4]`, that is $a[7] + a[5..6] + a[1..4]$, exactly $a[1..7]$, with three additions instead of seven. An update at position 3 walks $3 \rightarrow 4 \rightarrow 8 \rightarrow 16$: it changes `bit[3]`, `bit[4]` and `bit[8]`, precisely the blocks that contain position 3, and stops because 16 exceeds n . In binary the pattern is plain: a query clears the lowest 1-bit at each step, an update adds it, and both touch at most one block per bit of n .

Now combine this with Equation (8.2). When node 5 of **Figure 8.1** gains 10, the program adds +10 at position $tin[5] = 4$ and -10 at position $tout[5] + 1 = 8$. A query at position 6 (node 9) collects the +10 from `bit[4]` and does not reach position 8, so $S(9)$ grows by 10. A query at position 9 (node 6) collects both the +10 and the -10 through `bit[8]`, so $S(6)$ is unchanged, as it must be, since node 5 is not an ancestor of node 6.

8.2.5 Gear 3: binary lifting

Let $up[u][j]$ be the ancestor of u that is 2^j steps higher, or 0 if there is none. The table is filled from the relation

$$up[u][j] = up[up[u][j-1]][j-1] \quad (8.5)$$

because jumping 2^j steps is jumping 2^{j-1} steps twice. If the depth-first search fills the row of u when it enters u , all ancestors of u already have complete rows, so Equation (8.5) is always available. With 20 columns we can jump up to $2^{19} > 10^5$ steps at once.

Binary lifting answers *searches* along a path. Suppose a property P holds for v 's low ancestors and fails for the high ones, with one switch point in between: a *monotone* property. Then we can find the highest ancestor satisfying P by trying jumps of $2^{19}, 2^{18}, \dots, 2^0$ in that order and taking each jump that lands on a node where P still holds. It is binary search, with powers of two in place of midpoints, and it takes $O(\log n)$ jumps. The same jump pointers are the classic route to lowest common ancestors [3], and the competitive-programming literature uses them for every kind of upward search [4].

KEY INSIGHT: Compose by contracts. Ask: “If I could get $S(w)$ for any w in $O(\log n)$, what would the rest of the problem become?” Here it becomes a monotone search along an upward path, which binary lifting solves in $O(\log n)$ probes. Each probe calls the first contract once, so a question costs $O(\log^2 n)$.

8.3 Warm-Up: The Three Gears on a Small Tree

Listing 8.1 builds the tree of **Figure 8.1**, gives its nodes the values 5, 3, 2, 7, 1, 4, 6, 2, 8, prints entry times, exit times and root-to-node sums, jumps with binary lifting, and finally adds 10 to node 5.

Listing 8.1 `tree_gears.cpp`: Euler tour, Fenwick path sums and binary lifting on nine nodes.

```
// Euler tour + BIT (root-to-node path sums) and binary lifting on a small tree.
#include <iostream>
#include <vector>
using namespace std;
const int N = 9, LOG = 4;
vector<int> adj[N + 1];
int tin[N + 1], tout[N + 1], up[N + 1][LOG], dep[N + 1], timer_ = 0;
long long bit[N + 2];
void add(int i, long long v) {
    for (; i <= N; i += i & -i) bit[i] += v;
}
long long pref(int i) {
    long long s = 0;
    for (; i > 0; i -= i & -i) s += bit[i];
    return s;
}
void dfs(int u, int p) {
    tin[u] = ++timer_;
    up[u][0] = p;
    for (int j = 1; j < LOG; j++) up[u][j] = up[up[u][j - 1]][j - 1];
    for (int v : adj[u])
        if (v != p) dep[v] = dep[u] + 1, dfs(v, u);
    tout[u] = timer_;
}
void addNode(int u, long long y) { // node value += y
    add(tin[u], y);
    add(tout[u] + 1, -y);
}
int kth(int u, int k) { // k-th ancestor, 0 if it does not exist
    for (int j = 0; j < LOG; j++)
        if (k >> j & 1) u = up[u][j];
    return u;
}
int main() {
    int e[8][2] = {{1, 2}, {1, 3}, {2, 4}, {2, 5}, {3, 6}, {5, 7}, {5, 8}, {7, 9}};
    for (auto& x : e) adj[x[0]].push_back(x[1]), adj[x[1]].push_back(x[0]);
    dfs(1, 0);
    long long val[N + 1] = {0, 5, 3, 2, 7, 1, 4, 6, 2, 8};
    for (int u = 1; u <= N; u++) addNode(u, val[u]);
    cout << "node tin tout S(root->node)\n";
    for (int u = 1; u <= N; u++)
        cout << u << " " << tin[u] << " " << tout[u] << " " << pref(tin[u])
            << "\n";
    cout << "2nd ancestor of 9 = " << kth(9, 2) << ", 3rd = " << kth(9, 3) << "\n";
    addNode(5, 10); // node 5 gains 10: every path through 5 grows
    cout << "after +10 at node 5: S(9) = " << pref(tin[9])
```

Listing 8.1 (Continued) *tree_gears.cpp*: Euler tour, Fenwick path sums and binary lifting on nine nodes.

```
<< ", S(4) = " << pref(tin[4]) << "\n";
}
```

```
node tin tout S(root->node)
1 1 9 5
2 2 7 8
3 8 9 7
4 3 3 15
5 4 7 9
6 9 9 11
7 5 6 15
8 7 7 11
9 6 6 23
2nd ancestor of 9 = 5, 3rd = 2
after +10 at node 5: S(9) = 33, S(4) = 15
```

Check a few numbers by hand. The path from the root to node 9 is 1,2,5,7,9 with values $5 + 3 + 1 + 6 + 8 = 23$. The ancestors of 9 two and three steps up are 5 and 2. After adding 10 at node 5, $S(9)$ grows by 10 because 5 lies on its path, while $S(4)$ stays at 15, because node 4 is not in the subtree of 5: its entry time 3 is outside the interval $[4,7]$.

8.4 Case Study: HBD

8.4.1 The problem in our own words

The problem is SPOJ HBD [1]. The first line holds n and q , both up to 10^5 . The next $n - 1$ lines are the edges of a tree whose root is node 1. Then come n integers, the initial values of nodes $1..n$. Finally there are q operations of two kinds:

- **2 u y**: the value of node u grows by y .
- **1 u v x**: node u is an ancestor of v (possibly v itself). Walk upward from v toward u , adding the values of the nodes you visit, v included. Print how many nodes you have visited at the moment the total first reaches at least x . If even the whole path from v to u sums to less than x , print -1 .

Values and increments are non-negative. This matters: it makes the running total grow as we walk up, so “have we reached x yet?” is a monotone property, which is exactly what binary lifting needs.

8.4.2 Sample input and output

We use the tree of **Figure 8.1** with the same values as the warm-up:

```

9 6
1 2
1 3
2 4
2 5
3 6
5 7
5 8
7 9
5 3 2 7 1 4 6 2 8
1 1 9 10
1 2 9 16
1 5 9 100
2 7 5
1 5 9 16
1 1 4 20
```

```

2
4
-1
2
-1
```

Walking up from node 9 visits 9,7,5,2,1 with values 8,6,1,3,5 and running totals 8,14,15,18,23. The first question needs 10: reached at the second node. The second needs 16: reached at the fourth node (18), and node 2 is still inside the allowed path. The third needs 100 but the path from 9 to 5 sums to only 15. Then node 7 grows to 11, the totals become 8,19, ..., and 16 is reached at the second node. The last question walks 4,2,1 with total $7 + 3 + 5 = 15 < 20$.

8.4.3 Designing the solution

1. **One depth-first search** from the root computes tin , $tout$, the depth, and the lifting table by Equation (8.5). Node 0 is a sentinel: depth 0, and all its jumps lead back to 0.
2. **Initial values** are loaded with Equation (8.2); an operation $2 \ u \ y$ does the same with y .
3. **A question $1 \ u \ v \ x$** . First compute the whole path sum $S(v) - S(\text{parent}(u))$ by Equation (8.4). If it is below x , print -1 .
4. **Otherwise search**. Let $curr = v$. For $j = 19$ down to 0, look at $p = up[curr][j]$. The jump is allowed if p is not above u , that is, $\text{depth}(p) \geq \text{depth}(u)$. It is taken if the nodes strictly below p on the path still sum to less than x :

$$S(v) - S(p) < x \Leftrightarrow S(p) > S(v) - x \quad (8.6)$$

5. **The answer** is $\text{depth}(v) - \text{depth}(curr) + 1$, the number of nodes from v up to $curr$.

A natural alternative deserves a comment. One could binary search on the *number of steps* k , computing the k -th ancestor with the lifting table for every candidate k and testing its path sum. That is correct but costs $O(\log n)$ candidates, each with an $O(\log n)$ ancestor computation and an $O(\log n)$ Fenwick query: $O(\log^3 n)$ per question. Descending the powers of two directly, as step 4 does, merges the binary search and the ancestor computation into one loop and saves a whole logarithmic factor.

The sentinel node 0 deserves a sentence of its own. Every jump that would leave the tree through the root lands on node 0, whose own jumps all lead back to 0 and whose depth is 0. Because every real node has depth at least 1, the depth test in step 4 automatically refuses any jump to the sentinel, and `get_S(0)` returns 0, which is exactly the value Equation (8.4) needs when u is the root and its “parent” is the imaginary node above it. One extra node thus removes three special cases from the code: the root’s parent, jumps beyond the root, and the path sum of a path that ends at the root.

Why is $curr$ the node where the total first reaches x ? The search keeps the invariant “the nodes strictly below $curr$ sum to less than x ”, which is true at the start (nothing is below v on the path). It ends at the *highest* such node not above u . If $curr = u$, the whole path sums to at least x (checked in step 3) while everything strictly below u sums to less, so u is the stopping node. Otherwise the parent of $curr$ fails Equation (8.6): the nodes from v through $curr$ sum to at least x , while those strictly below $curr$ sum to less. Either way $curr$ is the first node at which the total reaches x .

8.4.4 Tracing one question

Take the second question of the sample, `1 2 9 16`, before any update. The depths along the path are 5 for node 9, 4 for node 7, 3 for node 5, 2 for node 2 and 1 for the root. The root-to-node sums are $S(9) = 23$, $S(7) = 15$, $S(5) = 9$, $S(2) = 8$ and $S(1) = 5$; the parent of $u = 2$ is the root, so the whole path from 9 up to 2 sums to $23 - 5 = 18 \geq 16$, and the search starts with $curr = 9$ and threshold $S(v) - x = 7$.

- For $j = 19$ down to $j = 3$, the jump leads beyond the root to the sentinel, whose depth 0 is below $\text{depth}(u) = 2$: refused.
- $j = 2$: four steps up from 9 is node 1, depth 1, above u : refused.
- $j = 1$: two steps up is node 5, depth 3, allowed. $S(5) = 9 > 7$, so the nodes strictly below node 5 (that is, 9 and 7) sum to $14 < 16$: jump, $curr = 5$.

- $j = 0$: one step up from 5 is node 2, depth 2, allowed. $S(2) = 8 > 7$, so the nodes strictly below node 2 (9, 7 and 5) sum to $15 < 16$: jump, $curr = 2$.

The answer is $\text{depth}(9) - \text{depth}(2) + 1 = 5 - 2 + 1 = 4$, matching the second output line. Notice how the jumps of 2 and then 1 together climb three levels: every number of steps from 0 to $2^{20} - 1$ is a sum of distinct powers of two, and the search picks exactly the binary digits of the distance to the stopping node.

8.4.5 The complete solution

Listing 8.2 is the full program, recorded as *Accepted* on SPOJ. The P50 block is the binary-lifting search of step 4.

Listing 8.2 [HBD.cpp](#): Euler tour, Fenwick tree and binary lifting composed.

```
#include <iostream>
#include <vector>
using namespace std;
const int MAXN = 100005;
vector<int> adj[MAXN];
int up[MAXN][20];
int depth[MAXN];
int in[MAXN], out[MAXN];
int timer_dfs = 0;
long long bit[MAXN];
int n, q;
// Fast manual input: one character at a time, no locking
inline long long read_ll() {
    long long x = 0; int f = 1; char ch = getchar_unlocked();
    while (ch < '0' || ch > '9') { if (ch == '-') f = -1; ch = getchar_unlocked(); }
    while (ch >= '0' && ch <= '9') { x = x * 10 + ch - '0'; ch = getchar_unlocked(); }
    return x * f;
}
inline void add(int idx, long long val) {
    for (; idx <= n; idx += idx & -idx) bit[idx] += val;
}
inline long long query(int idx) {
    long long sum = 0;
    for (; idx > 0; idx -= idx & -idx) sum += bit[idx];
    return sum;
}
inline long long get_S(int u) {
    if (u == 0) return 0;
    return query(in[u]);
}
void dfs(int u, int p, int d) {
    in[u] = ++timer_dfs;
    depth[u] = d;
    up[u][0] = p;
    for (int i = 1; i < 20; i++) {
        up[u][i] = up[up[u][i - 1]][i - 1];
    }
    for (int v : adj[u]) {
        if (v != p) {
            dfs(v, u, d + 1);
        }
    }
    out[u] = timer_dfs;
}
int main() {
    ios_base::sync_with_stdio(false);
```

Listing 8.2 (Continued) *HBD.cpp: Euler tour, Fenwick tree and binary lifting composed.*

```

cin.tie(NULL);
n = read_ll();
q = read_ll();
for (int i = 0; i < n - 1; i++) {
    int u = read_ll(), v = read_ll();
    adj[u].push_back(v);
    adj[v].push_back(u);
}
// Depth 0 is our dummy out-of-bounds node
depth[0] = 0;
dfs(1, 0, 1);
for (int i = 1; i <= n; i++) {
    long long c = read_ll();
    add(in[i], c);
    add(out[i] + 1, -c);
}
for (int i = 0; i < q; i++) {
    int type = read_ll();
    if (type == 1) {
        int u = read_ll(), v = read_ll();
        long long x = read_ll();
        long long S_v = get_S(v);
        long long S_parent_u = get_S(up[u][0]);
        // If the total sum on the path is not enough
        if (S_v - S_parent_u < x) {
            cout << "-1\n";
            continue;
        }
        // <P50>
        // Climb from v with jumps 2^19..2^0, never above u: jump to p while the
        // sum strictly below p, S_v - S_p, is still smaller than x.
        // >>> YOUR TURN (Public Edition): write this part yourself, following the hint
        // above. <<<
        // </P50>
        cout << depth[v] - depth[curr] + 1 << "\n";
    } else {
        int u = read_ll();
        long long y = read_ll();
        add(in[u], y);
        add(out[u] + 1, -y);
    }
}
return 0;
}

```

8.4.6 Line by line

Globals. `adj` holds the tree, `up` the lifting table with 20 columns, `depth`, `in` and `out` the Euler data, and `bit` the Fenwick array of `long long`: 10^5 values, each increased up to 10^5 times, can exceed the range of `int`.

read_ll. A hand-written reader with `getchar_unlocked`, skipping non-digits and remembering a minus sign. With about half a million numbers it saves noticeable time. The program never mixes it with `cin` for input; `cout` is used only for output.

add, query, get_S. The Fenwick operations of Section 8.2.2, and `get_S(u)`, which is $S(u)$ from Equation (8.3), returning 0 for the sentinel node 0.

dfs. It stamps *tin*, sets the depth and the direct parent, fills the rest of the row with Equation (8.5), visits the children and stamps *tout*. For the root, $p = 0$, so all its jumps go to the sentinel.

main, set-up. It reads the edges into adjacency lists, runs `dfs(1, 0, 1)` and loads the initial values with Equation (8.2).

A question. `S_v` and `S_parent_u` give the whole path sum; if it is too small, print `-1` and continue. The P50 block performs the search of step 4: the test `depth[p] >= depth[u]` refuses jumps above `u` (and to the sentinel, whose depth 0 is below every real depth), and the test `S_p > S_v - x` is Equation (8.6). The last line prints the node count.

An update. Two Fenwick additions, as in Equation (8.2).

8.4.7 Complexity and traps

The search makes 20 probes, each a Fenwick query of about 17 steps, so a question costs roughly 350 elementary steps and 10^5 questions about $3.5 \cdot 10^7$: comfortably fast. We measured the largest input we could build, a chain of 10^5 nodes (the deepest possible tree) with 10^5 mixed operations: 0.08 seconds.

PITFALL: Deep recursion. On a chain, `dfs` recurses 10^5 levels deep. That is fine on Linux and on the judge, whose stacks are large, but the default 1 MB stack of a Windows program can overflow and crash silently. On Windows compile with `-Wl,--stack=268435456`, or write the search iteratively (Exercise 5).

- **Off by one in the path sum.** The path from v to u includes u , so subtract $S(\text{parent}(u))$, not $S(u)$.
- **Jumping above u .** Without the depth check the search may count nodes that are not allowed.
- **Negative values.** If values could be negative, the running total would not be monotone and binary lifting would silently give wrong answers. Read the constraints before choosing a binary search.
- **int sums.** Totals exceed 2^{31} after enough updates; everything that holds a sum is `long long`.

8.5 Stress Testing Against a Brute Force

Listing 8.3 does what the organizer in Section 8.1 would do by hand: it stores parent pointers, applies updates directly to an array, and answers a question by walking up one node at a time.

Listing 8.3 `brute.cpp`: the walk-up reference for HBD.

```
// Reference for HBD: parent array, walk up node by node.
#include <cstdio>
#include <vector>
using namespace std;
int main() {
    int n, q;
    if (scanf("%d%d", &n, &q) != 2) 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);
    }
    vector<int> par(n + 1, 0), st = {1}, seen(n + 1, 0);
    seen[1] = 1;
    while (!st.empty()) {
        int u = st.back();
        st.pop_back();
        for (int v : adj[u])
            if (!seen[v]) seen[v] = 1, par[v] = u, st.push_back(v);
    }
    vector<long long> val(n + 1);
    for (int i = 1; i <= n; i++)
        if (scanf("%lld", &val[i]) != 1) return 0;
    while (q--) {
        int t;
        if (scanf("%d", &t) != 1) return 0;
        if (t == 1) {
            int u, v;
            long long x, s = 0;
            if (scanf("%d%d%lld", &u, &v, &x) != 3) return 0;
            int cnt = 0, w = v, ans = -1;
            while (true) {
                s += val[w], cnt++;
                if (s >= x) { ans = cnt; break; }
                if (w == u) break;
                w = par[w];
            }
            printf("%d\n", ans);
        } else {
            int u;
            long long y;
            if (scanf("%d%lld", &u, &y) != 2) return 0;
            val[u] += y;
        }
    }
}
```

Listing 8.3 (Continued) *brute.cpp: the walk-up reference for HBD.*

```
}
```

The generator, **Listing 8.4**, builds a random tree on up to 12 nodes by giving every node $v \geq 2$ a random parent below it, prints each edge in a random direction (so the program cannot rely on the order), and produces questions whose u is a genuine ancestor of v .

Listing 8.4 *gen.py*: random small trees and operations for HBD.

```
# Random small HBD inputs (values >= 0, x >= 1): python3 gen.py SEED
import random, sys
random.seed(int(sys.argv[1]))
n, q = random.randint(1, 12), random.randint(1, 15)
par = [0, 0] + [random.randint(1, i - 1) for i in range(2, n + 1)]
print(n, q)
for v in range(2, n + 1):
    print(par[v], v) if random.random() < .5 else print(v, par[v])
print(*[random.randint(0, 9) for _ in range(n)])
for _ in range(q):
    if random.random() < .7:
        v = random.randint(1, n)
        path = [v]
        while path[-1] != 1:
            path.append(par[path[-1]])
        print(1, random.choice(path), v, random.randint(1, 40))
    else:
        print(2, random.randint(1, n), random.randint(0, 9))
```

all 300 random tests passed

It helps to know which mistakes this test catches. Subtracting $S(u)$ instead of $S(\text{parent}(u))$ fails whenever a question's answer is exactly the whole path, which the generator produces often because its thresholds are small. A missing depth check lets the search climb above u and fails on questions whose u is not the root. Forgetting the second Fenwick addition of [Equation \(8.2\)](#) makes an update leak into nodes outside the subtree, and random edge directions catch programs that assume the first endpoint of each edge is the parent. What the small test cannot reveal is a deep-recursion crash or a slow reader; for those, the chain of 10^5 nodes from [Section 8.4.7](#) is the right companion test.

8.6 The Same Gears, Other Problems

The three gears reappear in many problems, and knowing the standard combinations saves a great deal of thinking during a contest.

Subtree sums. Store $val[u]$ at position $tin[u]$ of an ordinary Fenwick tree (point update, prefix query). Then the sum over the subtree of u is the range $[tin[u], tout[u]]$, two prefix queries by [Equation \(8.1\)](#). This is the mirror image of the trick used in HBD: there a node's value is spread over its subtree interval, here the interval is summed.

Adding to a whole subtree. “Increase every value in the subtree of u by y ” combined with “tell me the value of node w ” is, once more, [Equation \(8.2\)](#): the range update now describes the operation itself, and a point query reads the result.

Paths between any two nodes. Binary lifting also finds the lowest common ancestor of two nodes a and b : lift the deeper node to the same depth, then lift both together with the largest jumps that keep them apart. With the lowest common ancestor, any path splits into two vertical paths, and [Equation \(8.4\)](#) handles each (Exercise 4).

The k -th node on a path. Lifting by the binary digits of k , as the warm-up program does with `kth`, gives the k -th ancestor in $O(\log n)$; together with the lowest common ancestor it gives the k -th node on any path.

When a problem statement mentions a rooted tree, changing values and questions about “up” or “below”, run through this list before designing anything new. Very often the problem is one of these combinations with a small twist, and the twist is the only part that needs original thought.

8.7 Contest-Day Tactics

- **Write the contracts first.** On paper: “ $S(w)$ in $O(\log n)$ with updates”, “ k -th ancestor in $O(\log n)$ ”, “monotone search upward”. Then implement each gear separately and test it separately.
- **Test each gear alone.** Print tin , $tout$ and $S(w)$ for a small tree, as the warm-up program does, and compare with a hand drawing before writing the search. A bug found in a gear costs minutes; the same bug found through the whole solution costs an hour.
- **Recognize the identities.** Subtree equals Euler interval; root-to-node path equals a Fenwick prefix after Equation (8.2); vertical path equals a difference by Equation (8.4). These three sentences unlock a large family of tree problems.
- **Look for monotonicity before binary searching.** Non-negative values, non-decreasing depths, growing totals: name the reason your search is valid.
- **Build chain and star tests.** A chain exposes recursion depth and long jumps; a star exposes nodes with many children. Both take a minute to generate, and together with a random tree they cover the three shapes on which tree programs usually fail.
- **Budget the log factors.** $O(\log^2 n)$ per query is fine for 10^5 queries; for 10^6 it may not be, and then a different composition (for example walking the Fenwick tree directly) is needed [5].

8.8 Chapter Summary

- An Euler tour numbers the nodes so that every subtree is an interval, Equation (8.1).
- A Fenwick tree with range update and point query keeps root-to-node sums under changing values, Equation (8.2) and Equation (8.3); vertical path sums are differences, Equation (8.4).
- Binary lifting, Equation (8.5), turns a monotone search along an upward path into $O(\log n)$ jumps.
- HBD composes the three gears: each question costs $O(\log^2 n)$ and each update $O(\log n)$.

8.9 Exercises

1. Compute tin and $tout$ by hand for the tree with edges 1–2, 1–3, 3–4, 3–5, 5–6, visiting children in increasing order. Which interval is the subtree of 3?
2. Suppose values are stored on *edges* instead of nodes (each non-root node w carries the value of the edge to its parent). How do Equation (8.4) and the answer to a question change?
3. Give a tiny example with one negative value where the binary-lifting search returns a wrong answer, while the brute force is right.
4. How would you answer “sum of values on the path between any two nodes a and b ”, when neither is an ancestor of the other?
5. Rewrite `dfs` of Listing 8.2 without recursion.
6. How much memory does the lifting table of Listing 8.2 use, and how could it be halved?

8.10 Solutions to the Exercises

1. The order of entry is 1, 2, 3, 4, 5, 6, so tin is 1 to 6 in that order. Exit times: node 2 leaves at 2, node 4 at 4, node 6 at 6, node 5 at 6, node 3 at 6, node 1 at 6. The subtree of 3 is the interval $[3, 6]$, which holds nodes 3, 4, 5 and 6.
2. The path from v up to u now uses the edges of nodes $v, \dots, \text{child}(u)$, so its sum is $S(v) - S(u)$ rather than $S(v) - S(\text{parent}(u))$. A question counts edges instead of nodes, so the search must

- stay strictly below u (depth check with $>$), and the answer becomes $\text{depth}(v) - \text{depth}(\text{curr}) + 1$ edges, where curr is the lower end of the last edge used.
3. A chain $1 - 2 - 3$ rooted at 1 with values $\text{val}[3] = 5$, $\text{val}[2] = -5$, $\text{val}[1] = 5$, and the question **1 1 3 5**. Walking up from 3 the total is 5 at the first node, so the answer is 1. The search, however, tests the jump of two steps first: $S(1) = 5 > S(3) - 5 = 0$, so it jumps to node 1 and prints 3. The monotonicity is broken by the negative value.
4. Use the lowest common ancestor c of a and b (found with the same lifting table): the sum is $S(a) + S(b) - 2S(c) + \text{val}[c]$, because the ancestors of c are counted twice and c itself must be counted once.
5. Keep an explicit stack of pairs (node, index of the next child to visit). When a node is pushed, stamp tin , depth and its lifting row; when its child index runs past the end of its list, stamp tout and pop it. Skip the parent while scanning children. The order of visits, and so all the numbers, are the same as in the recursive version.
6. The table has $100\,005 \times 20$ values of type `int`, four bytes each: about 8 MB. Since $2^{17} = 131\,072 > 10^5$, 17 columns are enough, which saves 15 percent; a larger saving comes from storing only the jump of 2^j for the j given by the lowest set bit of the depth (the “jump pointer” layout), which needs one extra pointer per node at the cost of a slightly longer search. For HBD, 8 MB is well within the limit and the simple table is the right choice.

References

- [1] Sphere Online Judge, “Problem HBD.” [Online]. Available: <https://www.spoj.com/problems/HBD/>
- [2] P. M. Fenwick, “A new data structure for cumulative frequency tables,” *Software: Practice and Experience*, vol. 24, no. 3, pp. 327–336, 1994.
- [3] M. A. Bender and M. Farach-Colton, “The LCA problem revisited,” in *LATIN 2000: Theoretical Informatics*, LNCS vol. 1776. Berlin, Germany: Springer, 2000, pp. 88–94.
- [4] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Lulu Press, 2020.
- [5] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

Chapter 9

Counting Distinct Objects with Dynamic Programming — DSUBSEQ

“How many ways?” and “how many different ones?” are two of the most common questions at the olympiad, and they are not the same question. Counting ways is usually a straightforward dynamic program. Counting *different* objects hides a trap: the same object can be produced in several ways, and a careless recurrence counts it several times. This chapter is about avoiding that trap with a proof rather than with luck. You will learn to write a state definition as a full sentence, derive a recurrence from it, prove with a set argument that nothing is counted twice, and keep all arithmetic correct modulo a large prime. The case study, SPOJ DSUBSEQ [1], counts the distinct subsequences of strings of up to 10^5 letters, and its answer is one of the most elegant recurrences in the olympiad repertoire: *double the count, then subtract what the previous occurrence of the same letter already produced*.

9.1 Distinct Words from One Word

A teacher writes a word on the board, say BANANA, and asks the class to make new “words” by crossing out any letters they like while keeping the others in order. Soon the class realizes that many different crossings give the same result: crossing out letters 1, 3, 5 of BANANA gives ANA, and so does crossing out letters 1, 5, 6 and keeping 2, 3, 4. How many *different* words can the class possibly make?

There are $2^6 = 64$ ways to choose which letters to keep, but far fewer different results, because the repeated A’s and N’s make many choices coincide. For a word of 10^5 letters, the number of choices is 2^{100000} , a number with about thirty thousand digits, so listing is out of the question. We need a way to count the distinct results directly, and a way to be sure that the count is right, because nobody will be able to check a thirty-thousand-digit answer by hand.

A first idea is to generate the results one by one and keep them in a set, throwing away repeats. For BANANA this works instantly. For a string of forty letters it would already need more memory than any computer has, because the number of *different* results can itself be exponential: a string of forty different letters has 2^{40} , more than a trillion, distinct subsequences. So we must count without listing, and counting without listing is exactly where double counting sneaks in: we can no longer *see* the duplicates, we must *prove* where they are.

This question is not only a classroom game. In genetics, a DNA strand is a string over four letters and a subsequence models what survives after some positions are lost; in data compression and text indexing, counting distinct subsequences or substrings measures how much variety a text contains. And in olympiad problems the same “last occurrence” idea we are about to derive counts distinct subsets of a multiset, distinct labelled walks and distinct strings obtainable by deletions.

9.2 Counting Without Double Counting

9.2.1 Subsequences

A *subsequence* of a string $s = s_1 s_2 \dots s_n$ is what remains after deleting any set of positions, keeping the order of the rest. AGH is a subsequence of ABCDEFGH; AHG is not, because H comes after G in the original. The empty string, written ε , is a subsequence of every string (delete everything). We always count *strings*, not sets of positions: two different sets of positions that produce the same string count once.

9.2.2 The state, as a sentence

Every wrong counting program begins with a fuzzy state; the dynamic-programming method [2] is only as good as the state it is given. So before any formula we write one sentence:

DEFINITION. $dp[i]$ is the number of distinct subsequences of the prefix $s_1 s_2 \dots s_i$, the empty subsequence included. We write D_i for the set of those subsequences, so $|D_i| = dp[i]$, and $D_0 = \{\varepsilon\}$.

Make the sentence concrete at once. For the string AB, the prefix of length 0 is empty and $D_0 = \{\varepsilon\}$, so $dp[0] = 1$. The prefix A has $D_1 = \{\varepsilon, A\}$ and $dp[1] = 2$. The prefix AB has $D_2 = \{\varepsilon, A, B, AB\}$ and $dp[2] = 4$. Writing out the first two or three sets by hand, before any recurrence, is the cheapest way to discover that a definition is ambiguous: if you hesitate over whether ε belongs in D_0 , the sentence has not yet said enough. Everything that follows is a consequence of this sentence. If a later step seems to need something the sentence does not say, the sentence is wrong and must be fixed first.

9.2.3 Deriving the recurrence

Let $c = s_i$ be the new letter. A subsequence of the prefix of length i either does not use position i , and then it is a subsequence of the shorter prefix, or it does, and then it is a subsequence of the shorter prefix followed by c . So

$$D_i = D_{i-1} \cup \{t c : t \in D_{i-1}\} \quad (9.1)$$

The second set has exactly $|D_{i-1}|$ elements, because appending the same letter to different strings gives different strings. If the two sets in Equation (9.1) were disjoint, $dp[i]$ would be $2 dp[i-1]$. They are disjoint when c has never appeared before: then no string in D_{i-1} contains c , but every string in the second set ends with c .

When c has appeared before, let p be its most recent earlier position ($p < i$, $s_p = c$). The overlap of the two sets consists of the strings in D_{i-1} that end with c . Each of these can be formed with its final c taken at position p , the last c available, and its remaining letters from positions before p ; so the overlap is $\{t c : t \in D_{p-1}\}$, which has exactly $|D_{p-1}|$ elements. By inclusion-exclusion [3],

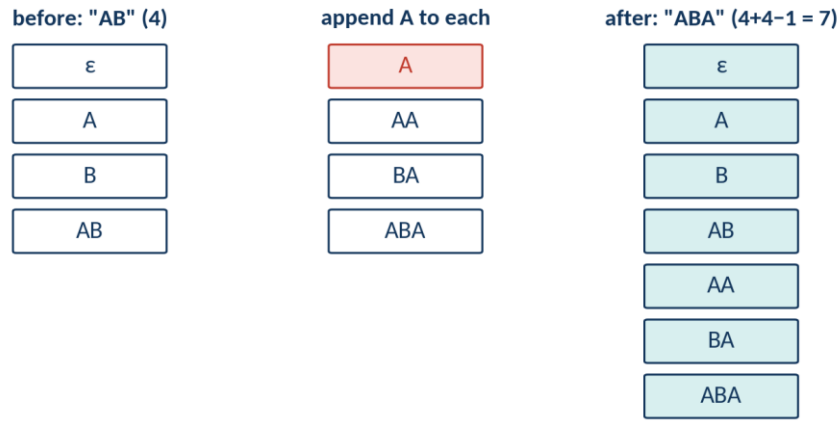
$$\begin{aligned} dp[i] &= 2 dp[i-1] - dp[p-1] \quad (\text{if } c \text{ appeared before at } p), & dp[i] \\ &= 2 dp[i-1] \quad (\text{otherwise}) \end{aligned} \quad (9.2)$$

with $dp[0] = 1$. The answer for the whole string is $dp[n]$.

KEY INSIGHT: The last occurrence owns the duplicates. Every duplicate created by the new c was already created when c appeared at position p , and exactly $dp[p-1]$ strings were created

then. Earlier occurrences of c need no correction of their own: their duplicates were already removed when p was processed.

Figure 9.1 shows the step from AB to ABA. The four old subsequences are kept, the four new ones are made by appending A, and exactly one of the new ones, the string A made from ϵ , was already there. Here $p = 1$, so the correction is $dp[0] = 1$ and the count is $4 + 4 - 1 = 7$.



"A" is produced twice: once long ago (ϵ + first A) and again now (ϵ + new A); subtract $dp[\text{last}-1] = dp[0] = 1$.

Figure 9.1 From AB to ABA: appending A to each old subsequence creates one duplicate, the string A, which the previous A already produced.

9.2.4 Two tempting mistakes

It is instructive to see what goes wrong without the proof. The first tempting recurrence is “double, and subtract one if the letter appeared before”, on the idea that only the single-letter string is repeated. On ABA it happens to work, giving $4 \cdot 2 - 1 = 7$. On ABAB it gives $7 \cdot 2 - 1 = 13$ instead of 12, because the second B re-creates *two* old strings, B and AB, as the table in Section 9.3.2 shows. The correction is not a constant: it is the number of strings that could precede the letter the previous time, $dp[p - 1]$.

The second tempting recurrence subtracts $dp[q - 1]$ for *every* earlier occurrence q of the letter, not only the last one. On AAA it gives $dp[3] = 2 \cdot 3 - dp[1] - dp[0] = 6 - 2 - 1 = 3$, but the correct answer is 4. The strings removed by the older occurrence were removed once already when the more recent one was processed; subtracting them again counts them as missing. The set argument of Section 9.2.3 prevents both mistakes: it names the overlap precisely, and the overlap is described by the last occurrence alone.

9.2.5 A second derivation: counting by the last letter

A different state gives the same numbers and is sometimes easier to extend. Let $e[c]$ be the number of distinct *non-empty* subsequences, seen so far, that end with the letter c . When the next letter of the string is c , the distinct subsequences that end with c are now exactly the strings tc where t is any distinct subsequence seen so far, the empty one included. So

$$e[c] \leftarrow 1 + \sum_d e[d], \quad \text{answer} = 1 + \sum_d e[d] \quad (9.3)$$

where the sums run over all 26 letters and the 1 stands for the empty string. The update *replaces* $e[c]$ rather than adding to it: the old strings ending in c are all among the new ones, which is how

the duplicates disappear. On ABAB, the values of $(e[A], e[B])$ are (1,0), (1,2), (4,2) and (4,7), and the answer is $1 + 4 + 7 = 12$, as before. Equation (9.3) costs 26 steps per letter instead of one, which is still fast, and it generalizes naturally to questions such as “how many distinct subsequences end with a vowel?”.

9.2.6 Arithmetic modulo a prime

The count grows like 2^n , so the problem asks for it modulo $M = 10^9 + 7$. Addition and multiplication commute with taking remainders, so we may reduce after every step [3]. Subtraction needs care: in C++, the `%` operator keeps the sign of the left operand, so $(a - b) \bmod M$ computed as $(a - b) \% M$ is negative whenever $a < b$, which happens all the time with reduced values. The safe form is

$$(a - b) \bmod M = ((a - b) + M) \bmod M \quad \text{for } 0 \leq a, b < M \quad (9.4)$$

The size of intermediate values matters too: $2 \cdot dp[i - 1] < 2 \cdot 10^9 + 14$ fits in a signed 64-bit integer with room to spare (and would *not* fit in a 32-bit `int`).

9.3 Warm-Up Problems

9.3.1 Counting ways: occurrences of ABC

First the easier cousin: count the *ways* a pattern occurs as a subsequence, where different sets of positions count separately. Let $f[j]$ be the number of ways to match the first j letters of the pattern using the text read so far. When the next text letter equals pattern letter j , every way to match the first $j - 1$ letters extends, so $f[j]$ grows by $f[j - 1]$. Updating j from high to low ensures that one text letter is used at most once per way, the same reason the 0/1 knapsack loop runs downward. Listing 9.1 counts ABC in four strings.

Listing 9.1 `abc_count.cpp`: counting occurrences of a pattern as a subsequence.

```
// How many times does "ABC" occur as a subsequence? (positions count, mod p)
#include <iostream>
#include <string>
using namespace std;
int main() {
    const long long MOD = 1000000007;
    string pat = "ABC";
    for (string s : {"ABC", "AABBCC", "ABCABC", "CBA"}) {
        // f[j] = number of ways to match the first j letters of pat so far
        long long f[4] = {1, 0, 0, 0};
        for (char ch : s)
            for (int j = 3; j >= 1; j--) // downward: each letter used once per way
                if (ch == pat[j - 1]) f[j] = (f[j] + f[j - 1]) % MOD;
        cout << s << ": " << f[3] << "\n";
    }
}
```

```
ABC: 1
AABBCC: 8
ABCABC: 4
CBA: 0
```

In AABBCC there are $2 \cdot 2 \cdot 2 = 8$ choices of one A, one B and one C, all in the right order. In ABCABC the four ways are the position triples (1,2,3), (1,2,6), (1,5,6) and (4,5,6). Here nothing is

deduplicated, because positions are what we count; the case study counts strings, and that is the whole difference.

9.3.2 Watching the recurrence work

Listing 9.2 prints every step of Equation (9.2) for ABAB, without the modulus so the numbers can be checked by hand.

Listing 9.2 `dsub_table.cpp`: the DSUBSEQ table for ABAB, step by step.

```
// Prints the DSUBSEQ table step by step for one string.
#include <iomanip>
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s = "ABAB";
    long long dp[5];
    int last[26] = {};
    dp[0] = 1;
    cout << " i char  2*dp[i-1]  minus  dp[i]\n";
    for (int i = 1; i <= 4; i++) {
        int c = s[i - 1] - 'A';
        long long twice = 2 * dp[i - 1], minus = last[c] ? dp[last[c] - 1] : 0;
        dp[i] = twice - minus;
        last[c] = i;
        cout << setw(2) << i << setw(5) << s[i - 1] << setw(11) << twice << setw(7)
            << minus << setw(7) << dp[i] << "\n";
    }
}
```

i	char	2*dp[i-1]	minus	dp[i]
1	A	2	0	2
2	B	4	0	4
3	A	8	1	7
4	B	14	2	12

At $i = 3$ the previous A is at position 1, so the correction is $dp[0] = 1$. At $i = 4$ the previous B is at position 2, so the correction is $dp[1] = 2$: the strings B and AB were produced by the first B and would be produced again. The twelve subsequences of ABAB are ϵ , A, B, AA, AB, BA, BB, AAB, ABA, ABB, BAB and ABAB; listing them is a worthwhile five-minute check.

9.3.3 BANANA, the classroom word

Back to the story of Section 9.1. For BANANA the recurrence gives, letter by letter: B is new, $dp[1] = 2$; A is new, $dp[2] = 4$; N is new, $dp[3] = 8$. The second A has its previous occurrence at position 2, so $dp[4] = 16 - dp[1] = 14$. The second N has its previous occurrence at position 3, so $dp[5] = 28 - dp[2] = 24$. The third A has its previous occurrence at position 4, so $dp[6] = 48 - dp[3] = 40$. Of the 64 ways to cross out letters, only 40 different words result, the empty word included; the solution program in Section 9.4 prints exactly 40 for this input. The class would need a long afternoon to list them all, and the recurrence needs six lines of arithmetic.

9.3.4 From ways to strings

Put the two warm-ups side by side, because the contrast is the lesson of the chapter. In **Listing 9.1** a way is a set of positions, and two different sets are two different ways even when they spell the same letters; there is nothing to deduplicate, and the recurrence simply adds. In **Listing 9.2** an object is a *string*, and the whole difficulty is that many sets of positions spell the

same string. The recurrence must therefore choose one *canonical* set of positions for each string and count only that one.

The last-occurrence rule is such a choice, although it is hidden in the proof. Among all the ways to spell a subsequence, pick the one that takes every letter as *late* as possible: the last letter at its last possible position, the letter before it at the last possible position before that, and so on. Every distinct string has exactly one such greedy spelling, so counting greedy spellings counts strings. When a letter c arrives at position i , the strings whose greedy spelling ends at i are all the strings $t c$; those that used to end at the previous c now move to i , and that move is precisely the subtraction of $dp[p - 1]$. Seeing a counting recurrence as “count one canonical representative per object” is the most reliable way to design new ones.

9.4 Case Study: DSUBSEQ

9.4.1 The problem in our own words

The problem is SPOJ DSUBSEQ [1]. The first line holds the number of test cases T , at most 100. Each test case is one string of uppercase letters A–Z, at most 10^5 long. For each string, print the number of distinct subsequences, *including the empty one*, modulo 1 000 000 007.

The phrase “including the empty one” deserves its own line. It shifts every answer by exactly one, so a program that excludes the empty subsequence fails every test. The official sample below settles the question: AAA has the distinct subsequences ε , A, AA and AAA, four in all.

9.4.2 Sample input and output

```
2
AAA
ABCDEFGG
```

```
4
128
```

ABCDEFGG has seven different letters, so no two choices of positions coincide and all $2^7 = 128$ choices are distinct: Equation (9.2) never subtracts. For AAA, the table is $dp = 1, 2, 3, 4$: every new A doubles and then subtracts the count from before the previous A.

9.4.3 Designing the solution

1. Keep `last[c]`, the most recent position of letter c , or 0 if c has not appeared. Reset it for every test case.
2. Set $dp[0] = 1$ and scan the string. For position i with letter c : compute $2 dp[i - 1] \bmod M$; if `last[c]` is not 0, subtract $dp[last[c] - 1]$ using Equation (9.4); then set `last[c] = i`.
3. Print $dp[n]$.

The order of the last two actions in step 2 matters: `last[c]` must be read *before* it is overwritten with the current position.

The array `dp` is kept whole only because the correction needs $dp[p - 1]$ for an old position p . A small observation removes even that: the only old values ever needed are $dp[last[c] - 1]$, one per letter. So we may store, for each letter, `before[c]`, the value of dp just before its most recent occurrence, and update it to the current $dp[i - 1]$ each time the letter appears. The whole algorithm then needs 26 numbers and one running value, independent of n . For DSUBSEQ the array is small enough that this refinement is unnecessary, but the same idea matters when the

strings are streamed and too long to store, and it is exactly what [Equation \(9.3\)](#) does in another form.

The cost is $O(n)$ per string and at most 10^7 steps for the whole input, plus reading 10^7 characters. Memory is one array of $10^5 + 2$ values of 64 bits, under a megabyte.

9.4.4 The complete solution

Listing 9.3 is the full program, recorded as *Accepted* on SPOJ. The P50 block holds the correction and the update of `last`.

Listing 9.3 `DSUBSEQ.cpp`: distinct subsequences with the last-occurrence correction.

```
#include <stdio>
#include <cstring>
using namespace std;
const int MOD=1000000007;
long long dp[100002];
int last[26];
int main() {
    int t; if(scanf("%d",&t)!=1) return 0;
    while(t--) {
        char s[100001]; if(scanf("%100000s",s+1)!=1) return 0;
        int n=strlen(s+1);
        memset(last,0,sizeof last);
        dp[0]=1;
        for(int i=1;i<=n;i++) {
            int c=s[i]-'A';
            dp[i]=2*dp[i-1]%MOD;
            // <P50>
            // If c was seen before at position last[c], remove the dp[last[c]-1]
            // subsequences counted twice (keep the value non-negative), then record i.
            // >>> YOUR TURN (Public Edition): write this part yourself, following the hint
            // above. <<<
            // </P50>
        }
        printf("%lld\n",dp[n]%MOD);
    }
}
```

9.4.5 Line by line

Globals. `MOD` is the prime $10^9 + 7$. `dp` is a global array of `long long`, large enough for the longest string plus position 0. `last` holds one position per letter.

Reading. `scanf("%d", &t)` reads the number of tests; each string is read into `s + 1` so that positions run from 1 to n , matching the mathematics. The width limit `%100000s` guarantees that a longer line can never overflow the buffer, and checking the return value stops cleanly at the end of the input. The buffer lives inside the loop on the stack; at about 100 KB it is well within any default stack.

Per test. `memset` clears `last`, `dp[0] = 1` is the empty subsequence, and the loop applies [Equation \(9.2\)](#): first the doubling, reduced modulo M , then (in the P50 block) the correction when `last[c]` is set, written as [Equation \(9.4\)](#), and finally the update of `last[c]`.

Reading the input is a noticeable part of the running time: up to 10^7 characters. `scanf("%s")` is fast enough here, taking a few hundredths of a second, because it copies a whole word at a time. Reading character by character with `cin >> ch` would be several times slower, and `getline` into

a `std::string` followed by a copy is also fine. When input sizes grow another factor of ten, the block reader of Chapter 7 is the safe choice.

Output. $dp[n]$ is already reduced; the extra `% MOD` is harmless.

9.4.6 Why the modulus is harmless

A modulus might seem to endanger the subtraction in Equation (9.2): the true value $2 dp[i - 1] - dp[p - 1]$ is always positive, because the doubled set contains the overlap, but after reduction the stored numbers are only remainders, and the stored $2 dp[i - 1]$ may be smaller than the stored $dp[p - 1]$. That is not an error in the mathematics. Remainders respect addition, subtraction and multiplication, so the remainder of the true answer equals the remainder of the computation carried out on remainders, provided the result is brought back into the range $0..M - 1$. Equation (9.4) does exactly that. The choice of $M = 10^9 + 7$ is conventional: it is prime, it is just below 2^{30} , so a sum of two remainders fits in a 32-bit signed integer and a product of two fits in a 64-bit one. Primality does not matter for this problem, which never divides, but it matters in many counting problems that need modular inverses, so the same modulus is used everywhere.

9.4.7 Traps

PITFALL: Writing `(dp[i] - dp[last[c]-1]) % MOD`. When the reduced value of `dp[i]` happens to be smaller than the subtracted one, the result is negative, and every later value inherits the error. The fix is one `+ MOD`, as in Equation (9.4).

- **Empty or not.** Read the statement for the words “empty” or “non-empty”. Here the empty subsequence counts, so the answer is $dp[n]$, not $dp[n] - 1$.
- **Forgetting to reset `last`.** Positions from the previous string would make the first letters of the next string subtract corrections that do not exist.
- **Subtracting for every earlier occurrence.** Only the most recent one counts; the key insight above explains why.
- **A 32-bit `dp`.** $2 dp[i - 1]$ can reach about $2 \cdot 10^9$, beyond the `int` range.

9.5 Stress Testing Against a Brute Force

For short strings we can literally list every subsequence. Listing 9.4 tries all 2^n choices of positions, builds each string and puts it into a `std::set`, which removes duplicates for us; the answer is the size of the set.

Listing 9.4 `brute.cpp`: listing every subsequence into a set.

```
// Reference for DSUBSEQ: put every subsequence (as a string) into a set.
#include <iostream>
#include <set>
#include <string>
using namespace std;
int main() {
    int t;
    cin >> t;
    while (t--) {
        string s;
        cin >> s;
        int n = s.size();
        set<string> all;
        for (int mask = 0; mask < (1 << n); mask++) {
            string sub;
            for (int i = 0; i < n; i++)
                if (mask >> i & 1) sub += s[i];
            all.insert(sub); // the empty string is included (mask 0)
        }
        cout << all.size() << "\n";
    }
}
```

The generator in **Listing 9.5** writes up to six strings of length up to 14 over alphabets of one to four letters. Small alphabets matter: with 26 letters, random short strings rarely repeat a letter and the correction in [Equation \(9.2\)](#) would almost never be tested.

Listing 9.5 `gen.py`: random short strings over small alphabets.

```
# Random small DSUBSEQ inputs: python3 gen.py SEED
import random, sys
random.seed(int(sys.argv[1]))
T = random.randint(1, 6)
print(T)
for _ in range(T):
    k = random.randint(1, 4)
    print(''.join(random.choice('ABCD'[:k]) for _ in range(random.randint(1, 14))))
```

all 300 random tests passed

It is worth thinking about *which* bugs this test would catch. A program that forgets the empty string prints every answer one too small and fails the first comparison. A program that subtracts a constant, or subtracts for every earlier occurrence, fails as soon as a letter appears three times or two letters interleave, which happens in almost every generated string over two or three letters. A program that forgets to reset `last` between test cases fails on the second string of the first file with more than one string. A missing `+ MOD` is the one bug this test *cannot* catch, because the numbers for strings of 14 letters never reach the modulus. For that bug the right test is different: a long string, for example 5000 letters, compared against a second, independent implementation such as [Equation \(9.3\)](#), where both programs reduce modulo M . Knowing the limits of a test is as important as running it.

The largest legal input, 100 strings of 10^5 random letters, took about 0.11 seconds.

9.6 The Pattern in Other Problems

The idea “the last occurrence owns the duplicates” is a pattern, not a trick for one problem. Three relatives appear regularly.

Distinct subsets of a multiset by sum. When items with equal values are indistinguishable, a subset-sum count per value must treat a group of k equal items as “take 0, 1, ..., k of them”, not as k

separate yes-or-no choices; otherwise the same multiset is counted several times. The grouping plays the role that the last occurrence plays here: it assigns every object to exactly one way of producing it.

Distinct strings after deletions with a rule. Problems such as “how many different strings can be obtained by deleting exactly k characters” or “by deleting characters so that no two equal letters are adjacent” are solved by adding the extra information to the state, exactly as Exercise 4 adds the length. The correction term keeps its shape: subtract what the previous occurrence already produced *with the same extra information*.

Counting distinct substrings. Contiguous pieces are a different story, because a substring is fixed by its start and end rather than by a free choice of positions. SPOJ’s DISUBSTR, the subject of Chapter 13, counts them with a suffix structure instead; the danger of double counting is the same, and the remedy again is to assign each distinct object to a single canonical way of producing it.

The general recipe is therefore: (1) describe the objects as a set; (2) describe how the set grows when the input grows by one element, as in Equation (9.1); (3) name the overlap exactly and count it; (4) only then write code. Steps (1) to (3) fit on half a page and turn a guess into a proof. When step (3) is hard, look for a canonical representative, as in Section 9.3.4: if every object can be produced in exactly one agreed way, counting those ways is counting the objects, and the overlap never has to be computed at all.

9.7 Contest-Day Tactics

- **Write the state sentence first** [4]. “dp of i is the number of distinct ... of the first i ..., empty included.” If you cannot finish the sentence, you are not ready for the recurrence.
- **Prove distinctness with sets.** Write the set equation such as Equation (9.1), identify the overlap and count it. Five lines of proof save an hour of debugging.
- **Make + MOD a reflex.** Every subtraction under a modulus gets it, even when you believe the values are ordered.
- **Enumerate for $n \leq 12$.** A brute-force enumeration into a set takes three minutes to write and catches off-by-one errors in the empty string, the first occurrence and the modulus.
- **Use tiny alphabets in tests.** Repetition is where counting bugs live; generate it on purpose.
- **Cross-check two derivations.** When two different states, such as Equation (9.2) and Equation (9.3), give the same numbers on random inputs, both are very likely right; this catches the bugs that small brute forces cannot reach.

9.8 Chapter Summary

- Counting distinct objects differs from counting ways: duplicates must be proved absent, not hoped absent.
- The state definition is a full sentence; the set equation Equation (9.1) follows from it.
- For distinct subsequences, the overlap created by a letter c is exactly what its previous occurrence created, giving Equation (9.2): double, then subtract $dp[p - 1]$.
- Counting distinct objects means counting one canonical representative per object; for subsequences the canonical spelling takes every letter as late as possible.
- Modular subtraction needs $+M$, Equation (9.4); intermediate values need 64 bits.

- Short strings over small alphabets, enumerated into a set, are the right brute force; a second derivation such as Equation (9.3) covers the long strings.

9.9 Exercises

1. Compute dp for ZZZZ by hand. What is the answer for a string of n equal letters, and why?
2. How does the answer change if the empty subsequence must *not* be counted? Change one line of Listing 9.3.
3. Prove by induction on i that Equation (9.2) gives $\$D_i\$$ for every i .
4. Extend the method to count the distinct subsequences of length exactly k . Give the recurrence.
5. The string may now contain lowercase and uppercase letters and digits. What must change in the program?
6. Use Equation (9.3) to compute the answer for BANANA and compare it with Section 9.3.3.
7. For a string of length n over an alphabet of two letters, what is the largest possible answer? Try small n with the brute force and guess the pattern.

9.10 Solutions to the Exercises

1. $dp[0..4] = 1, 2, 3, 4, 5$. At each step the doubling adds all old strings with one more Z, and the correction removes all of them except the longest, because every shorter string of Z's already existed. The distinct subsequences of n equal letters are the strings of $0, 1, \dots, n$ letters, so the answer is $n + 1$.
2. Print $(dp[n] - 1 + MOD) \% MOD$. The recurrence itself must keep counting the empty string, because the empty string is what makes single letters appear as new subsequences.
3. Base: $D_0 = \{\epsilon\}$ and $dp[0] = 1$. Step: assume $\$j\} = D_j\$$ for all $j < i$. Equation (9.1) writes D_i as a union of D_{i-1} and a set of the same size. If $c = s_i$ is new, the sets are disjoint (only the second contains c), so $|D_i| = 2|D_{i-1}| = 2 dp[i - 1]$. Otherwise the overlap is $\{t c : t \in D_{p-1}\}$ as shown in Section 9.2.3, of size $|D_{p-1}| = dp[p - 1]$ by the hypothesis, and inclusion-exclusion gives Equation (9.2).
4. Let $dp[i][k]$ count the distinct subsequences of length k of the prefix of length i . The same set argument, restricted to length k , gives $dp[i][k] = dp[i - 1][k] + dp[i - 1][k - 1] - dp[p - 1][k - 1]$ (the last term only if c appeared before at p), with $dp[i][0] = 1$. The cost is $O(nk)$.
5. Only the alphabet: make `last` large enough for every character code, for example `int last[128]`, and index it with `(unsigned char)s[i]` instead of `s[i] - 'A'`. The recurrence does not care what the letters are.
6. With $(e[B], e[A], e[N])$: after B, (1,0,0); after A, (1,2,0); after N, (1,2,4); after A, $e[A] = 1 + 7 = 8$, giving (1,8,4); after N, $e[N] = 1 + 13 = 14$, giving (1,8,14); after A, $e[A] = 1 + 23 = 24$, giving (1,24,14). The answer is $1 + 1 + 24 + 14 = 40$, the same as the recurrence.
7. Experiments with the brute force for $n \leq 12$ show that the alternating strings ABAB... and BABA... give the most. For them each new letter's previous occurrence is two positions back, so $dp[i] = 2 dp[i - 1] - dp[i - 3]$ with $dp[0..2] = 1, 2, 4$; this produces 1, 2, 4, 7, 12, 20, 33, ..., which are Fibonacci numbers minus one: $dp[n] = F_{n+3} - 1$ with $F_1 = F_2 = 1$. Intuitively, a letter equal to its predecessor makes the correction as large as possible ($dp[i - 2]$), and alternation avoids that at every single step.

References

- [1] Sphere Online Judge, “Problem DSUBSEQ.” [Online]. Available: <https://www.spoj.com/problems/DSUBSEQ/>
- [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. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1994.
- [4] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Lulu Press, 2020.

Chapter 10

Bitmask and Profile Dynamic Programming — CFONISMG

When a problem says “at most 16 people” or “at most 4 columns”, the jury is sending a message: an exponential algorithm is welcome, as long as it is a *controlled* exponential. This chapter teaches the two standard ways to control it. *Bitmask dynamic programming* stores a set of up to twenty objects as the bits of one integer, so that a set becomes an array index. *Profile dynamic programming* fills a grid cell by cell and remembers only the thin boundary between the finished and the unfinished part, the *profile*, instead of the whole grid. You will learn to encode both, to count how many states they create before writing any code, and to keep only the states that can actually occur. The case study, SPOJ CFONISMG [1], seats students in a classroom so that friends sit together and troublemakers sit apart; its state combines a bitmask of the students already seated with a profile of the last row of seats, and it is solved by a map that holds only the reachable states.

10.1 Seating a Classroom

A teacher prepares the seating plan for a small classroom with R rows of C desks. She has a list of students who may be seated; not every desk has to be used, and nobody can sit in two places. She knows which pairs of students are friends: a pair of friends sitting side by side or one directly behind the other works well together, and she counts one point for every such pair. She also knows which pairs are rivals, who must never sit next to each other, and which students are disruptive; two disruptive students must never sit next to each other either. Which plan gives the most points?

Try to solve it by hand for a 2×3 room and four students, and you quickly realize why computers are needed. Each desk can be empty or hold any unused student, so there are hundreds of plans already; for a 4×4 room and sixteen students the number of plans has more than twenty digits. Yet the teacher’s reasoning has a useful shape. When she decides who sits at a desk, only two neighbors matter at that moment: the desk in front and the desk to the left, if she fills the room from the front row to the back and from left to right. And the only thing she must remember about the past, beyond those neighbors, is *who has already been seated*. Both pieces of memory are small, and that is the whole idea of this chapter.

The same shape of problem appears in tiling floors with rectangular tiles, in planting a field where some crops must not grow next to each other, in placing antennas on a grid so that neighbors do not interfere, and in the travelling-salesman problem for a delivery route through a dozen customers. In each, a small set or a narrow strip is all the past you need.

The chapter follows the two ideas in turn. Section 10.2 introduces bitmasks from zero, counts the cost of enumerating subsets and works the travelling-salesman problem as the model bitmask program. Section 10.3 introduces the profile, counts domino tilings with it, shows how to pack a profile into one number and, most importantly, how to estimate and measure the number of states before trusting an exponential program. Section 10.4 combines both ideas on CFONISMG,

with a hand-traced run, the complete program and its traps, and Section 10.5 tests it against a plain backtracking search.

10.2 Sets as Numbers

10.2.1 Bit operations

Number the objects $0, 1, \dots, n - 1$. A subset S is stored as the integer whose bit i is 1 exactly when object i belongs to S . The subset $\{0, 2, 3\}$ becomes $2^0 + 2^2 + 2^3 = 13$, binary 1101. Every subset of n objects is thus an integer between 0 and $2^n - 1$, and set operations become single machine instructions, as **Table 10.1** shows.

Table 10.1 Set operations on bitmasks; the last loop visits every non-empty subset of m .

Set operation	C++ expression
is i in S ?	<code>(mask >> i) & 1</code>
add i	<code>mask = (1 << i)</code>
remove i	<code>mask & ~(1 << i)</code>
union, intersection	<code>a = b, a & b</code>
size of S	<code>__builtin_popcount(mask)</code>
smallest element	<code>__builtin_ctz(mask)</code>
all subsets of m	<code>for (s = m; s; s = (s - 1) & m)</code>

Work one example in full. With $mask = 13$ (binary 1101, the set $\{0, 2, 3\}$): testing object 2 computes $13 \gg 2 = 3$, binary 11, whose last bit is 1, so 2 is a member; adding object 1 gives $13 | 2 = 15$, binary 1111; removing object 3 gives 13 with bit 3 cleared, 5, binary 0101; the size is $\text{popcount}(13) = 3$ and the smallest element is $\text{ctz}(13) = 0$, the number of trailing zero bits. Each of these is one or two machine instructions, which is why bitmask programs are so fast in practice.

The last row deserves a word. Subtracting 1 from s turns its lowest 1-bit into 0 and all lower bits into 1; the `& m` then keeps only the bits allowed by m . The result is the next smaller subset of m in numeric order, so the loop visits all non-empty subsets of m once each. Running it for every m costs, in total,

$$\sum_m 2^{\text{popcount}(m)} = \sum_{k=0}^n \binom{n}{k} 2^k = 3^n \quad (10.1)$$

steps, by the binomial theorem. For $n = 16$ that is about $4.3 \cdot 10^7$, fast; the naive “every m , every s , test whether $s \subseteq m$ ” costs $4^n \approx 4.3 \cdot 10^9$, far too slow.

10.2.2 The model: the travelling salesman

The classic bitmask dynamic program is Held and Karp’s solution of the travelling-salesman problem [2]. A salesman starts in city 0, must visit each of n cities exactly once and return. Let $dp[S][j]$ be the length of the shortest route that starts in city 0, visits exactly the cities of S and ends in city $j \in S$. The last step of such a route came from some city $i \in S$, $i \neq j$, so

$$dp[S][j] = \min_{i \in S, i \neq j} (dp[S \setminus \{j\}][i] + d(i, j)), \quad dp[\{0\}][0] = 0 \quad (10.2)$$

and the answer is $\min_j (dp[\text{all}][j] + d(j, 0))$. There are $2^n n$ states and each takes $O(n)$ work: $O(2^n n^2)$ in total, about $1.7 \cdot 10^7$ for $n = 16$, compared with $15! \approx 1.3 \cdot 10^{12}$ routes. The key observation is that the future of a partial route depends only on *which* cities are done and *where* we stand, not on the order in which the done cities were visited.

KEY INSIGHT: A bitmask state forgets the order of the past and remembers only its content. That is precisely the right amount of memory whenever the future depends only on “which ones are used”, and that is why it turns factorials into powers of two.

Listing 10.1 implements Equation (10.2) in “push” form: from each reached state it extends the route to every unvisited city.

Listing 10.1 `tsp_bitmask.cpp`: the Held–Karp dynamic program for five cities.

```
// Bitmask DP for the shortest round trip through all cities (Held-Karp).
#include <algorithm>
#include <iostream>
using namespace std;
int main() {
    const int n = 5, INF = 1e9;
    int d[n][n] = {{0, 3, 4, 2, 7},
                  {3, 0, 4, 6, 3},
                  {4, 4, 0, 5, 8},
                  {2, 6, 5, 0, 6},
                  {7, 3, 8, 6, 0}};

    static int dp[1 << n][n];
    for (auto& row : dp) fill(row, row + n, INF);
    dp[1][0] = 0; // visited {0}, standing at city 0
    for (int mask = 1; mask < (1 << n); mask++) {
        for (int last = 0; last < n; last++) {
            if (dp[mask][last] == INF) continue;
            for (int nxt = 0; nxt < n; nxt++) {
                if (mask >> nxt & 1) continue; // already visited
                int nm = mask | 1 << nxt;
                dp[nm][nxt] = min(dp[nm][nxt], dp[mask][last] + d[last][nxt]);
            }
        }
    }
    int best = INF;
    for (int last = 1; last < n; last++)
        best = min(best, dp[(1 << n) - 1][last] + d[last][0]);
    cout << "shortest tour = " << best << "\n";
}
```

shortest tour = 19

The tour $0 \rightarrow 3 \rightarrow 4 \rightarrow 1 \rightarrow 2 \rightarrow 0$ has length $2 + 6 + 3 + 4 + 4 = 19$. The masks are processed in increasing numeric order, which is valid because adding a city always makes the mask larger: every state is final before it is used. The same table could be filled in “pull” form, computing each $dp[S][j]$ directly from Equation (10.2) by looking at all its predecessors; both forms do the same work. The push form is often more convenient when many states are unreachable, because it never touches them, and that observation leads straight to the hash maps of the case study.

10.3 The Profile: a Grid's Cross-Section as State

10.3.1 Filling cell by cell

Number the cells of an $R \times C$ grid in reading order: cell k is in row $\lfloor k/C \rfloor$ and column $k \bmod C$. Suppose every constraint of the problem involves only neighbors that share a side. When we decide the content of cell k , its neighbor above is cell $k - C$ and its neighbor to the left is cell $k - 1$; the neighbors below and to the right are still empty and will check cell k themselves when their turn comes.

So how much of the filled part must be remembered? Exactly the cells that some *future* cell will look at. Cell k and every later cell look back at most C positions, so the future depends only on the last C filled cells, $k - C, \dots, k - 1$. This window is the *profile*. **Figure 10.1** shows it for $C = 4$ while cell 6 is being filled: cells 2, 3, 4, 5 form the profile, cell 2 is the neighbor above, cell 5 the neighbor to the left, and cells 0 and 1 will never be looked at again.

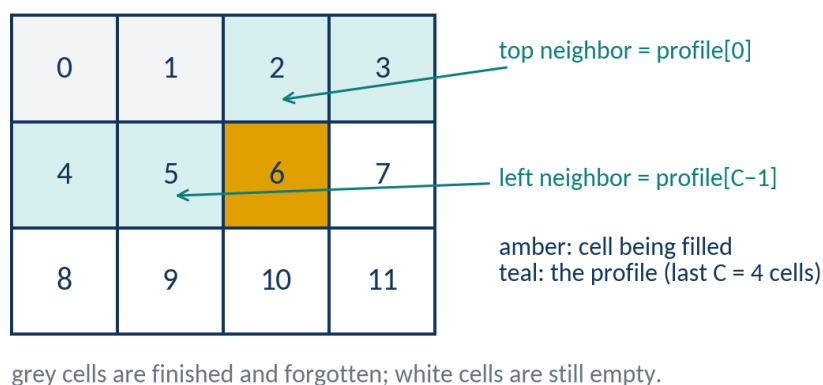


Figure 10.1 Filling a 3×4 grid in reading order: while cell 6 is filled, only the last four cells (the profile) can still influence the future.

DEFINITION. In a *profile dynamic program* over a grid filled in reading order, the state after k cells consists of the contents of cells $k - C, \dots, k - 1$ (the profile), plus whatever global information the problem needs, such as the set of objects already used.

10.3.2 Warm-up: domino tilings

The classic example counts the ways to cover an $R \times C$ board with 1×2 dominoes. Here each profile position needs only one bit: “this cell is already covered by a domino that sticks out from an earlier cell”. When we reach cell k in column c :

- if bit c is set, the cell is already covered from above; clear the bit and move on;
- otherwise place a vertical domino, covering this cell and the one below it, which sets bit c for the next row;
- or, if the cell to the right exists and is free, place a horizontal domino, which sets bit $c + 1$ for the current row.

The state has 2^C values and each step costs $O(1)$ per state, so the whole count costs $O(RC 2^C)$.

Listing 10.2 runs it for four boards.

Listing 10.2 `domino_profile.cpp`: counting domino tilings with a one-bit-per-column profile.

```
// Broken-profile DP: number of ways to tile an R x C board with 1x2 dominoes.
#include <iostream>
#include <vector>
using namespace std;
long long tilings(int R, int C) { // C <= 10: profile = C bits
    vector<long long> dp(1 << C, 0), nd;
    dp[0] = 1;
    for (int cell = 0; cell < R * C; cell++) {
        int c = cell % C;
        nd.assign(1 << C, 0);
        for (int m = 0; m < (1 << C); m++) {
            if (!dp[m]) continue;
            if (m >> c & 1) { // already covered from above: just move on
                nd[m & ~(1 << c)] += dp[m];
                continue;
            }
            nd[m | 1 << c] += dp[m]; // vertical domino sticks into next row
            if (c + 1 < C && !(m >> (c + 1) & 1))
                nd[m | 1 << (c + 1)] += dp[m]; // horizontal domino covers c+1
        }
        dp = nd;
    }
    return dp[0];
}
int main() {
    cout << "2x3: " << tilings(2, 3) << "\n";
    cout << "3x4: " << tilings(3, 4) << "\n";
    cout << "4x4: " << tilings(4, 4) << "\n";
    cout << "8x8: " << tilings(8, 8) << "\n";
}
```

```
2x3: 3
3x4: 11
4x4: 36
8x8: 12988816
```

The 2×3 board has three tilings (three vertical, or one vertical and two stacked horizontal ones on either side), and the chessboard count 12 988 816 is a well-known number, so this output doubles as a test. The broken-profile technique is treated at length in the standard handbooks [3], [4]. Notice that one bit per cell was enough here, because the only fact the future needs is “covered or not”. In CFONISMG the future needs to know *who* sits in a cell, so each profile position holds a whole student number.

10.3.3 Encoding a profile as one number

If a profile position can hold B different values, a profile of C positions is a number in base B :

$$prof = \sum_{k=0}^{C-1} profile[k] \cdot B^k \quad (10.3)$$

With $profile[0]$ the oldest cell (the one above the current cell) as the lowest digit, advancing to the next cell is simple arithmetic: dropping the oldest cell is “subtract digit 0, divide by B ”, and appending the new cell’s value s as the newest digit is “add $s \cdot B^{C-1}$ ”. For CFONISMG, $B = 17$ (0 for an empty desk, 1 to 16 for a student) and $C \leq 4$, so a profile is below $17^4 = 83\,521 < 2^{17}$.

10.3.4 Estimate before you code

Before writing a profile DP, count its states. The honest bound for CFONISMG multiplies the used-student masks, 2^{16} , by the profiles, 17^4 : about $5.5 \cdot 10^9$, hopeless as an array. But most of these

combinations can never occur: a student in the profile must also be in the mask, no student can appear twice in the profile, and after k cells at most k students are seated. That is why the solution stores states in a hash map, keyed by the packed pair (mask, profile), and creates an entry only when a state is actually reached.

How many are reached depends on the input, so we measured it. On random classrooms with about a third of all pairs friendly and a quarter rival, the largest number of states alive at one time was 33 350 for a 3×4 room with 8 students (0.03 seconds), 39 296 for 4×4 with 8 students (0.05 seconds), but 9.5 million for a 2×4 room with 16 students (almost 10 seconds). The profile is never the problem; the mask is. When many students compete for many desks, the number of possible “who has been seated” sets explodes. The program is exact for every size, and it was accepted by the judge, whose tests therefore keep this product manageable, but the measurement tells you exactly where the danger lies. Make the same measurement for every exponential solution you write; one extra line that prints the size of the map after each cell, run on the largest input and deleted before submitting, is all it takes.

10.3.5 Arrays or hash maps?

A profile DP can store its states in two ways. When every combination of mask and profile is possible and the product is small, as in the domino tiling with its 2^C states, a plain array indexed by the state is fastest: no hashing, perfect memory locality. When the theoretical product is huge but the reachable part is small, as in CFONISMG, a hash map such as `std::unordered_map` stores only what exists. The price is a constant factor: each lookup hashes the key and may follow a pointer, typically ten to fifty times slower than an array access. A middle road, useful when the map becomes the bottleneck, is to collect all new states of a cell in a vector of (key, score) pairs, sort it by key, and merge equal keys with a single pass; sorting has excellent locality and often beats hashing for millions of entries.

A second practical point is the *rolling* structure. The program never needs the states of two cells ago, so it keeps only two maps, the current one and the next, and swaps them with `std::move`, which transfers the internal storage without copying a single entry. The same rolling idea appears in almost every grid DP: memory proportional to one layer, not to the whole grid.

10.4 Case Study: CFONISMG

10.4.1 The problem in our own words

The problem is SPOJ CFONISMG [1]. The first line holds R , C and N : the classroom has R rows of C desks, $C \leq 4$, and there are N candidate students, at most 16. The next N lines each give a student’s name and a status word; the word **DISRUPTIVE** marks a disruptive student, any other word a calm one. Then comes a number F followed by F pairs of names who are friends, and a number V followed by V pairs of names who are rivals.

A seating plan puts each student at no more than one desk and leaves the other desks empty. Two desks are *adjacent* when they share a side. A plan is valid when no two adjacent desks hold rivals and no two adjacent desks hold two disruptive students. Its score is the number of adjacent pairs of desks holding friends. Print the largest score of a valid plan (the empty plan is valid, so the answer is at least 0).

10.4.2 Sample input and output

```
2 3 4
Ani CALM
Budi DISRUPTIVE
Citra CALM
Dedi DISRUPTIVE
3
Ani Budi
Ani Citra
Citra Dedi
1
Budi Citra
```

3

There are three friendly pairs, so 3 is the most one could hope for. It is reachable: seat Budi, Ani and Citra in the front row, in that order, and Dedi behind Citra. Ani sits next to both her friends, Citra has Dedi behind her, the rivals Budi and Citra are two desks apart, and the two disruptive students Budi and Dedi are not adjacent.

10.4.3 Designing the solution

- 1. Read and index.** Give the students numbers 1 to N through a map from names; record `disruptive[i]`, and the symmetric tables `is_friend_pair` and `is_rival`.
- 2. State.** After k cells, a state is a pair (mask of seated students, profile of the last C cells as in Equation (10.3)), and its value is the best score reachable with that state. Pack the pair into one 64-bit key: the mask shifted left by 17 bits, plus the profile.
- 3. Transition.** For the next cell, read the neighbor above (`profile[0]`, if the row is not the first) and the neighbor to the left (`profile[C - 1]`, if the column is not the first). Try $s = 0$ (leave the desk empty) and every student $s = 1..N$. Reject s if it is already in the mask, or if it is a rival of a neighbor, or if both s and a neighbor are disruptive. Otherwise the gain is the number of neighbors who are friends of s , and the new state is (mask with s added, profile shifted with s appended). Keep the maximum score per new key:

$$ndp[key'] = \max(ndp[key'], dp[key] + gain(s)) \quad (10.4)$$

- 4. Answer.** After all RC cells, the largest value in the map.

Why is keeping only the maximum per key correct? Two partial plans with the same key have seated the same students and have the same last C cells. Every future decision, its legality and its gain, depends only on those two things. So any completion of one plan is also a completion of the other, with the same extra score; the plan with the higher score so far can never be worse. This is the optimal-substructure argument behind every dynamic program [5], written for this state.

10.4.4 A tiny run by hand

Follow the program on the smallest interesting room: one row of three desks ($R = 1$, $C = 3$), a calm student Ani (number 1) and a disruptive student Budi (number 2), who are friends. Write a state as (seated set, profile), with the profile listed from oldest to newest desk.

Before any desk: one state, ($\emptyset$, $[0,0,0]$) with score 0.

Desk 0. From the single state the program tries the empty desk, Ani and Budi. There are no neighbors (row 0, column 0), so all three are legal and gain nothing: ($\emptyset$, $[0,0,0]$), ($\{1\}$, $[0,0,1]$) and ($\{2\}$, $[0,0,2]$), all with score 0.

Desk 1. The left neighbor is the newest profile digit. From $(\emptyset, \dots)$ the program can again leave the desk empty or seat either student: three states with score 0. From $(\{1\}, [0,0,1])$ it can leave the desk empty or seat Budi next to Ani, a friend: score 1, state $(\{1,2\}, [0,1,2])$. Symmetrically from $(\{2\}, [0,0,2])$ it can seat Ani next to Budi: score 1, state $(\{1,2\}, [0,2,1])$. Seven states in all; note that $(\{1\}, [0,1,0])$ (Ani on desk 0, desk 1 empty) and $(\{1\}, [0,0,1])$ (Ani on desk 1) are *different* keys, because Ani's position still matters for desk 2.

Desk 2. The same reasoning continues; for example $(\{1\}, [0,0,1])$ can seat Budi on desk 2 for a score of 1. After the last desk the largest score in the map is 1, which is correct: there is only one pair of friends. Tracing a run like this once, with pencil and paper, is the fastest way to become confident about which digit is “top” and which is “left”.

10.4.5 The complete solution

Listing 10.3 is the complete program, recorded as *Accepted* on SPOJ. The P50 block holds the scoring and the construction of the new key inside `try_place`.

Listing 10.3 `CFONISMG.cpp`: bitmask of seated students plus a base-17 profile, stored in a hash map.

```
#include <bits/stdc++.h>
using namespace std;
int R,C,N,cells;
bool disruptive[17],is_rival[17][17],is_friend_pair[17][17];
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cin>>R>>C>>N;
    cells=R*C;
    map<string,int> id;
    for(int i=1; i<=N; i++) {
        string nm,st;
        cin>>nm>>st;
        id[nm]=i;
        disruptive[i]=(st=="DISRUPTIVE");
    }
    int F;
    cin>>F;
    for(int i=0; i<F; i++) {
        string a,b;
        cin>>a>>b;
        int ia=id[a],ib=id[b];
        is_friend_pair[ia][ib]=is_friend_pair[ib][ia]=true;
    }
    int V;
    cin>>V;
    for(int i=0; i<V; i++) {
        string a,b;
        cin>>a>>b;
        int ia=id[a],ib=id[b];
        is_rival[ia][ib]=is_rival[ib][ia]=true;
    }
    int pw[5]= {1,17,289,4913,83521};
    const int PB=17;
    unordered_map<long long,int> dp;
    dp.reserve(1<<18);
    dp[0]=0;
    for(int cell=0; cell<cells; cell++) {
        int row=cell/C,col=cell%C;
        unordered_map<long long,int> ndp;
        ndp.reserve(dp.size()*4);
        for(auto&[key,score]:dp) {
            int bm=(int)(key>>PB),prof=(int)(key&((1<<PB)-1));
            int profile[4];
            {
                int v=prof;
```

Listing 10.3 (Continued) *CFONISMG.cpp: bitmask of seated students plus a base-17 profile, stored in a hash map.*

```

        for(int i=0; i<C; i++) {
            profile[i]=v%17;
            v/=17;
        }
    }
    int top_stu=(row>0)?profile[0]:0, left_stu=(col>0)?profile[C-1]:0;
    int base_prof=(prof-profile[0])/17;
    auto try_place=[&](int s) {
        if(s>0&&(bm&(1<<(s-1))))return;
        if(s>0) {
            if(top_stu>0&&(is_rival[s][top_stu]||
                (disruptive[s]&&disruptive[top_stu])))return;
            if(left_stu>0&&(is_rival[s][left_stu]||
                (disruptive[s]&&disruptive[left_stu])))return;
        }
        // <P50>
        // Score +1 per friend among top/left, add s to the bitmask, append s
        // to base_prof (oldest digit already dropped), keep the best score.
        // >>> YOUR TURN (Public Edition): write this part yourself, following the hint
above. <<<
        // </P50>
    };
    try_place(0);
    for(int s=1; s<=N; s++)try_place(s);
}
dp=move(ndp);
}
int ans=0;
for(auto&[k,v]:dp)ans=max(ans,v);
cout<<ans<<"\n";
}

```

10.4.6 Line by line

Tables. `disruptive`, `is_rival` and `is_friend_pair` are indexed by student numbers 1 to 16, so arrays of size 17 suffice. The friend and rival tables are filled in both directions, because adjacency has no direction.

Names. `map<string,int> id` turns names into numbers once, while reading; afterwards the program works only with numbers.

Powers and packing. `pw` holds $17^0, \dots, 17^4$; `PB = 17` is the number of bits reserved for the profile inside the key. The map starts with the single state “nobody seated, all desks empty”, key 0, score 0. `reserve` avoids repeated rehashing as the map grows.

The cell loop. For each cell the program computes its row and column and builds a fresh map `ndp`. For every stored state it unpacks the mask `bm` and the profile `prof`, decodes the profile digits with Equation (10.3), reads `top_stu` and `left_stu`, and precomputes `base_prof`, the profile with its oldest digit removed.

`try_place(s)`, first half. It rejects a student already in the mask and a student who clashes with the top or left neighbor, exactly as in step 3 of the design. The empty desk, $s = 0$, skips all checks.

`try_place(s)`, second half (the P50 block). It counts friends among the two neighbors, builds the new mask `nb` and the new key `nk` by appending s as the newest profile digit, and applies Equation (10.4) with one lookup in `ndp`.

End of the cell. `dp = move(ndp)` hands the new map over without copying it. After the last cell the program scans the map for the largest score and prints it.

10.4.7 Traps

PITFALL: Treating the left neighbor of a desk in column 0 as `profile[C - 1]`. That digit is the *last desk of the previous row*, which is not adjacent. The test `col > 0` is what prevents a wrong clash or a wrong friendship across the end of a row.

- **Too many students for the digits.** Base 17 allows 16 students. With 17 students, digit 17 would carry into the next position and corrupt the profile, and the arrays of size 17 would be indexed out of bounds.
- **An `int` key.** The key needs 17 bits of mask plus 17 bits of profile, 34 bits: it must be `long long`.
- **Forgetting the empty desk.** Without `try_place(0)` every desk must be filled, which is impossible when there are more desks than students, and the map empties.
- **Scoring a pair twice.** Each adjacent pair is scored exactly once, by the later of its two desks, looking back up or left. Looking in all four directions would double every score.
- **Trusting the state count blindly.** Measure it, as in Section 10.3.4, on the largest input you think the statement allows.

10.5 Stress Testing Against a Brute Force

The brute force in **Listing 10.4** tries every plan: for each desk in reading order it leaves the desk empty or seats any unused student who does not clash with the neighbors above and to the left. It explores the same decisions as the dynamic program, but without merging any states, so it is exponentially slower and completely independent of the key packing and the profile arithmetic.

Listing 10.4 `brute.cpp`: backtracking over every seating plan.

```
// Reference for CFONISMG: try every seating (empty or unused student per cell).
#include <iostream>
#include <map>
#include <string>
#include <vector>
using namespace std;
int R, C, N, best = 0;
bool dis[18], riv[18][18], fr[18][18], used[18];
vector<int> seat;
void go(int cell, int score) {
    if (cell == R * C) {
        best = max(best, score);
        return;
    }
    int r = cell / C, c = cell % C;
    int top = r > 0 ? seat[cell - C] : 0, left = c > 0 ? seat[cell - 1] : 0;
    seat[cell] = 0;
    go(cell + 1, score);
    for (int s = 1; s <= N; s++) {
        if (used[s]) continue;
        bool ok = true;
        int g = 0;
        for (int nb : {top, left}) {
            if (!nb) continue;
            if (riv[s][nb] || (dis[s] && dis[nb])) ok = false;
            if (fr[s][nb]) g++;
        }
        if (!ok) continue;
        used[s] = true, seat[cell] = s;
        go(cell + 1, score + g);
        used[s] = false, seat[cell] = 0;
    }
}
int main() {
    cin >> R >> C >> N;
    map<string, int> id;
    for (int i = 1; i <= N; i++) {
        string nm, st;
        cin >> nm >> st;
        id[nm] = i, dis[i] = (st == "DISRUPTIVE");
    }
    int F, V;
    cin >> F;
    for (int i = 0; i < F; i++) {
        string a, b;
        cin >> a >> b;
```

Listing 10.4 (Continued) *brute.cpp: backtracking over every seating plan.*

```
        fr[id[a]][id[b]] = fr[id[b]][id[a]] = true;
    }
    cin >> V;
    for (int i = 0; i < V; i++) {
        string a, b;
        cin >> a >> b;
        riv[id[a]][id[b]] = riv[id[b]][id[a]] = true;
    }
    seat.assign(R * C, 0);
    go(0, 0);
    cout << best << "\n";
}
```

The generator in **Listing 10.5** produces rooms of at most seven desks, up to five students, random statuses, and a random split of the pairs into friends, rivals and neutral pairs, so that no pair is both friendly and hostile.

Listing 10.5 `gen.py`: random small classrooms for CFONISMG.

```
# Random small CFONISMG inputs: python3 gen.py SEED
import random, sys, itertools
random.seed(int(sys.argv[1]))
while True:
    R, C = random.randint(1, 3), random.randint(1, 4)
    if R * C <= 7:
        break
N = random.randint(1, 5)
names = ['S%d' % i for i in range(1, N + 1)]
print(R, C, N)
for s in names:
    print(s, random.choice(['DISRUPTIVE', 'CALM', 'CALM']))
pairs = list(itertools.combinations(names, 2))
random.shuffle(pairs)
f = random.randint(0, len(pairs))
v = random.randint(0, len(pairs) - f)
print(f)
for a, b in pairs[:f]:
    print(a, b)
print(v)
for a, b in pairs[f:f + v]:
    print(a, b)
```

all 300 random tests passed

10.6 Contest-Day Tactics

- **Read the small limits as hints.** $N \leq 20$ suggests a bitmask over objects; a grid side ≤ 5 or so suggests a profile over that side. Both at once suggest both, as here.
- **Write the state count on paper.** Multiply the ranges, then subtract what cannot happen. If the product is too large even after that, use a map of reached states and *measure* it.
- **Fill in reading order and look only back.** Up and left are the only neighbors that exist when a cell is decided; each constraint and each score is then handled exactly once.
- **Pack keys carefully.** Decide the number of bits for each part, check that the sum fits in 64 bits, and write the pack and unpack code next to each other.
- **Use 64-bit masks when n reaches 31.** Shifting a 32-bit `int` by 31 positions gives a negative number and by 32 or more is undefined behavior in C++; `1LL << i` avoids both.
- **Keep a slow twin.** A backtracking brute force is the most natural stress-test partner for a state-merging dynamic program: same decisions, no merging.

10.7 Chapter Summary

- A bitmask stores a set of up to about 20 objects in one integer; set operations are single instructions, and enumerating all subsets of all masks costs 3^n , Equation (10.1).
- Held–Karp, Equation (10.2), shows the power of forgetting order: $O(2^n n^2)$ instead of $n!$.
- A profile DP fills a grid in reading order and remembers only the last C cells, encoded as a base- B number, Equation (10.3).
- CFONISMG combines both: (mask of seated students, profile of the last C desks), transitions by Equation (10.4), stored in a hash map of reached states.
- Exponential solutions must be *measured*: the reached-state count, not the theoretical bound, decides the running time.

10.8 Exercises

1. List the values of s produced by `for (s = m; s; s = (s - 1) & m)` for $m = 13$ (binary 1101).
2. Verify by hand that a 2×3 board has exactly three domino tilings, and describe them.
3. In **Listing 10.1**, what changes if the salesman does not need to return to city 0?
4. Modify the scoring of CFONISMG so that friends sitting *diagonally* behind each other (up-left) also score a point. What extra information must the profile contain?
5. Explain why the brute force in **Listing 10.4** and the dynamic program agree even though the dynamic program discards all but one plan per key.
6. In the tiny run of Section 10.4.4, how many states exist after desk 2 at most, and why are there fewer in reality?

10.9 Solutions to the Exercises

1. 13 (1101), 12 (1100), 9 (1001), 8 (1000), 5 (0101), 4 (0100), 1 (0001), and then 0 ends the loop: all seven non-empty subsets of $\{0,2,3\}$, in decreasing order.
2. Number the columns 1 to 3. Either all three dominoes are vertical; or column 1 holds a vertical domino and columns 2–3 hold two horizontal dominoes stacked on top of each other; or columns 1–2 hold two stacked horizontal dominoes and column 3 a vertical one. No other arrangement covers the board, so the count is 3, matching the program.
3. The answer becomes $\min_j dp[all][j]$ without the term $d(j, 0)$; the table itself is unchanged.
4. The up-left neighbor of cell k is cell $k - C - 1$, one position older than the current profile reaches. The profile must therefore hold $C + 1$ cells, and the up-left neighbor exists only when both the row and the column are positive. The base-17 encoding still fits: $17^5 \approx 1.4 \cdot 10^6 < 2^{21}$, so the profile needs 21 bits of the key.
5. Section 10.4.3 shows that two partial plans with the same key have exactly the same set of valid completions with the same extra scores. Discarding the worse one therefore never removes the best complete plan: its prefix either is the kept plan or has a score no higher than the kept plan with the same key, and the kept plan admits the same completion. The brute force examines every plan explicitly, so both find the same maximum.
6. Each state after desk 2 is determined by the contents of the three desks (the profile then covers the whole room), and each desk holds nobody, Ani or Budi, with no student twice: $1 + 2 \cdot 3 + 2 \cdot 3 = 13$ arrangements (none seated; one student on one of three desks; both students on two of three desks in either order). Budi and Ani next to each other are legal because they are friends, not rivals, and only one of them is disruptive, so all 13 can occur; the mask adds no new states here because it is determined by the profile. In larger rooms the mask remembers students who have left the profile, and that is where the growth measured in Section 10.3.4 comes from.

References

- [1] Sphere Online Judge, “Problem CFONISMG.” [Online]. Available: <https://www.spoj.com/problems/CFONISMG/>
- [2] M. Held and R. M. Karp, “A dynamic programming approach to sequencing problems,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 10, no. 1, pp. 196–210, 1962.
- [3] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Lulu Press, 2020.
- [4] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

- [5] 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.

Chapter 11

CDQ Divide and Conquer and the Two-Dimensional LIS — LIS2

The longest increasing subsequence is one of the first dynamic programs every contestant meets. Give each element two numbers instead of one, demand that *both* grow, and keep the input order, and the friendly $O(n \log n)$ trick stops working: the plain dynamic program needs $O(n^2)$ time, far too slow for $n = 10^5$. This chapter teaches the method that rescues it, *CDQ divide and conquer*: split the sequence by position, finish the left half, let the finished left half push its influence into the right half with a sweep and a Fenwick tree, and only then finish the right half. By the end you will be able to explain why that order is compulsory, prove the $O(n \log^2 n)$ bound, write the whole solution to the SPOJ problem LIS2 from memory, and check it against a brute force on hundreds of random inputs. The same pattern solves a whole family of problems in which one state depends on earlier states through two inequalities, which makes it one of the most valuable tools a contestant can carry into the national round.

11.1 Where This Helps: Dewi's Training Log

Dewi runs three or four times a week and writes every run in a notebook: the distance in kilometers and her average speed. After a year she wants to tell a simple story of progress. She will pick some runs from the notebook, in the order they happened, such that each chosen run is both *longer* and *faster* than the one chosen before it. How long can such a story be?

She cannot reorder the runs, because a story of progress must move forward in time. She cannot skip the “both” either: a run that was longer but slower does not show that she improved, and neither does a faster but shorter one. What she is asking for is the *longest chain* of runs that increases in two measurements at once while respecting the order of the dates.

The same question hides in many places. A shop owner looks for the longest stretch of months, not necessarily consecutive, in which both revenue and number of customers rose. A durian farmer who measures seedlings in the order they were planted looks for the longest sequence of trees, each taller and each with a thicker trunk than the previous one, to see whether a new fertilizer keeps helping. A scheduler of nested boxes, envelopes or tasks asks the same thing with sizes. In all of these the data arrive in a fixed order and each item carries a pair (x, y) .

With a few dozen runs Dewi could compare every pair by hand. With a hundred thousand records, as in the contest problem of this chapter, the obvious method compares about $5 \cdot 10^9$ pairs, which is far beyond the one or two seconds a judge allows. We need a method that is almost linear, and it will come from a divide-and-conquer idea that Chinese olympiad training made popular under the name *CDQ* [1].

11.2 The Technique from Zero

11.2.1 From one dimension to two

Start with one number per element. Given $a_1, a_2, \dots, a_n$, the *longest increasing subsequence* (LIS) is the longest list of positions $i_1 < i_2 < \dots < i_k$ with $a_{i_1} < a_{i_2} < \dots < a_{i_k}$. Let dp_i be the length of the longest such list that *ends* at position i . Every list ending at i is either i alone or a list ending at some earlier j with $a_j < a_i$, followed by i . So

$$dp_i = 1 + \max(\{0\} \cup \{dp_j : j < i, a_j < a_i\}) \quad (11.1)$$

and the answer is the largest dp_i . Equation (11.1) has two conditions on j : it must come *earlier* ($j < i$) and it must be *smaller* ($a_j < a_i$). The first condition is handled for free by processing positions from left to right; when we reach i , only earlier positions have been processed. The second is a *prefix maximum* over values: among all processed elements with value below a_i , what is the largest dp ? A Fenwick tree (binary indexed tree) that stores maxima instead of sums answers that in $O(\log n)$ [2], [3]. That gives the familiar $O(n \log n)$ LIS.

Now give each element a pair (x_i, y_i) and ask for a chain in which both coordinates increase strictly. The recurrence keeps its shape:

$$dp_i = 1 + \max(\{0\} \cup \{dp_j : j < i, x_j < x_i, y_j < y_i\}) \quad (11.2)$$

but Equation (11.2) has *three* conditions on j : earlier in the input, smaller in x and smaller in y . Processing left to right removes one of them, a Fenwick tree over y removes another, and the condition on x is left over with nothing to handle it. Sorting by x instead of by position does not help, because then the position condition is left over. Three independent orders cannot be handled by a single sweep plus a single tree. We need to remove one order by some other means.

11.2.2 The idea: split by position

CDQ removes the position order by divide and conquer. Cut the index range $[l, r]$ at its middle m . Every pair (j, i) with $j < i$ in the range is of one of three kinds:

- both in the left half $[l, m]$,
- both in the right half $[m + 1, r]$,
- j in the left half and i in the right half.

The first two kinds are the same problem on a smaller range, so recursion handles them. The third kind is the new part, and it has a lovely property: *every* left element comes earlier than *every* right element, so the position condition $j < i$ is automatically true for all these pairs. Only $x_j < x_i$ and $y_j < y_i$ remain, and two conditions are exactly what one sweep plus one Fenwick tree can handle. Sort the left half and the right half by x . Walk through the right elements in increasing x ; before handling a right element i , insert into the Fenwick tree every left element j with $x_j < x_i$, at position y_j with value dp_j . Then the largest value among Fenwick positions below y_i is exactly the best dp_j over all left elements that are allowed to precede i . Figure 11.1 draws both steps on a small example.

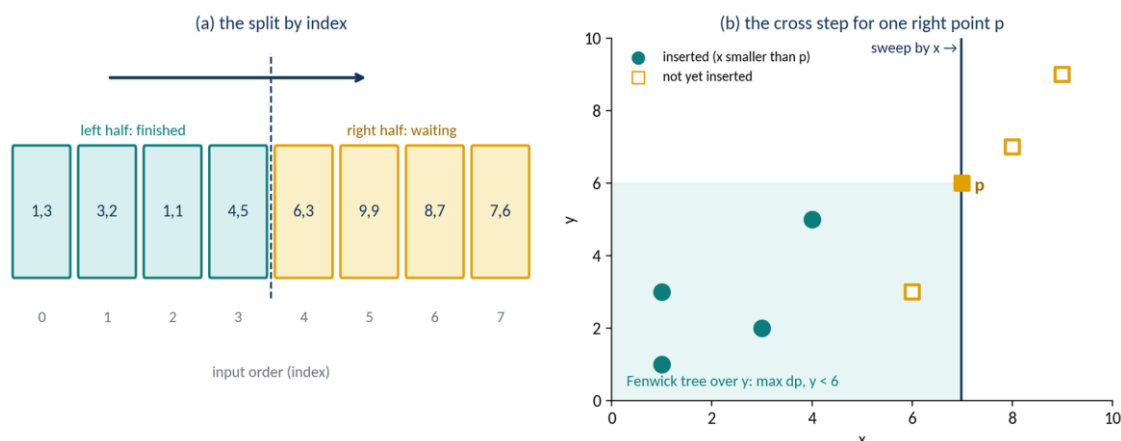


Figure 11.1 CDQ on eight points. (a) The range is cut by index; the left half is finished before it influences the right half. (b) For the right point $p = (7,6)$ every left point with a smaller x has been inserted into the Fenwick tree, which then answers “largest dp with $y < 6$ ”.

We call this middle step *cross*. The whole method fits in a few lines of pseudocode, shown in **Pseudocode 11.1**.

Pseudocode 11.1 CDQ divide and conquer for the two-dimensional LIS.

```
SOLVE(l, r)
1  if l >= r
2      return                                     // one element: dp is already 1
3  m = floor((l + r) / 2)
4  SOLVE(l, m)                                   // the left half becomes final
5  CROSS(l, m, r)                                // left dp values improve right dp values
6  SOLVE(m + 1, r)                              // the right half finishes

CROSS(l, m, r)
1  A = indices l..m sorted by x; B = indices m+1..r sorted by x
2  j = 1
3  for each i in B, in increasing x
4      while j <= |A| and x[A[j]] < x[i]
5          FENWICK-MAX-UPDATE(y[A[j]], dp[A[j]]); j = j + 1
6      dp[i] = max(dp[i], FENWICK-PREFIX-MAX(y[i] - 1) + 1)
7  undo the Fenwick positions written in line 5
```

DEFINITION. *CDQ divide and conquer* is an offline technique for recurrences in which an element’s value depends on earlier elements through extra order conditions. It splits the elements by time (position), solves the earlier half completely, passes the contribution of the earlier half to the later half in one batch (the *cross* step), and then solves the later half. The cross step no longer needs to check time, which removes one dimension from the problem [1], [4].

11.2.3 Why the order left, cross, right is compulsory

Look again at lines 4–6 of SOLVE. Two other orders look just as natural, and both are wrong.

If we called CROSS before SOLVE(l, m), the Fenwick tree would receive left values that are still incomplete: a left element that should reach 3 by way of other left elements might still hold 1. The right elements would then inherit a value that is too small.

If we called SOLVE($m + 1, r$) before CROSS, the right half would pass values among its own elements before those values had received the left influence. Suppose the best chain is $j \rightarrow i \rightarrow k$ with j in the left half and i, k in the right half. The link $j \rightarrow i$ is found in CROSS, and the link $i \rightarrow k$ is

found inside $\text{SOLVE}(m + 1, r)$. If the right half is solved first, k reads dp_i while it is still 1, and the chain is lost.

Here is the argument that the correct order computes every dp_i exactly, written as an invariant.

KEY INSIGHT. *Claim:* when $\text{SOLVE}(l, r)$ is called, every dp_i with $i \in [l, r]$ already includes the best contribution of every $j < l$; when it returns, every dp_i in $[l, r]$ is final. *Proof by induction on the length of the range.* A range of length 1 has no inner pairs, so its value is final on entry. For a longer range, $\text{SOLVE}(l, m)$ sees the same precondition and, by induction, returns with the left half final. CROSS then adds, for every right i , the best contribution of every $j \in [l, m]$, and it may do so in one batch precisely because those values are final. Now every right element carries the contribution of all $j \leq m$, which is the precondition of $\text{SOLVE}(m + 1, r)$, so by induction the right half comes back final.

The invariant also explains why the bug is so dangerous in a contest: on tiny inputs the chains are short, and many random tests have no chain of length 3 crossing the middle, so a program with the calls in the wrong order passes the samples and fails on the judge.

11.2.4 The cost

Let $T(n)$ be the time for a range of n elements. The cross step sorts both halves by x , which costs $O(n \log n)$, and performs at most n Fenwick updates, n queries and n clean-ups, each $O(\log n)$. So

$$T(n) = 2T(n/2) + O(n \log n) \quad (11.3)$$

The recursion tree of Equation (11.3) has $\log_2 n$ levels, and every level does $O(n \log n)$ work in total, because the ranges on one level partition the n elements. Hence $T(n) = O(n \log^2 n)$, by the same recursion-tree argument used for merge sort [5]. For $n = 10^5$ that is about $10^5 \cdot 17 \cdot 17 \approx 3 \cdot 10^7$ basic steps: a fraction of a second. The quadratic dynamic program of Equation (11.2), in contrast, needs about $5 \cdot 10^9$ comparisons.

One detail keeps that bound honest. After a cross step the Fenwick tree must be empty again, or the next cross step would see stale values. Clearing the whole array would cost $O(m)$ each time, where m is the number of distinct y values, and there are about n cross steps, so the clean-up alone could become $O(n \cdot m) = O(n^2)$. Instead we walk the same update paths again and write 0, which costs exactly as much as the updates did. The phrase to remember is: *undo only what you wrote*.

11.3 Warm-Up Problems

11.3.1 Inversions: the left half influences the right half

The simplest problem with the CDQ shape counts *inversions*: pairs $i < j$ with $a_i > a_j$. Split in the middle; inversions inside each half are counted recursively; inversions with i on the left and j on the right are counted while merging the two sorted halves. Whenever the merge takes an element from the right half, all left elements that are still waiting are larger, and each forms an inversion with it. Listing 11.1 is the complete program.

Listing 11.1 `inversions.cpp`: counting inversions with merge sort, the ancestor of CDQ.

```
// Warm-up: count pairs  $i < j$  with  $a[i] > a[j]$  by divide and conquer.
// The left half is finished before it "influences" the right half.
#include <cstdio>
#include <vector>
using namespace std;

long long sortCount(vector<int>& a, int l, int r) { // sorts  $a[l..r]$ 
    if (l >= r) return 0;
    int mid = (l + r) / 2;
    long long cnt = sortCount(a, l, mid) + sortCount(a, mid + 1, r);
    vector<int> merged;
    int i = l, j = mid + 1;
    while (i <= mid || j <= r) {
        if (j > r || (i <= mid && a[i] <= a[j])) {
            merged.push_back(a[i++]);
        } else {
            cnt += mid - i + 1; //  $a[j]$  is smaller than all of  $a[i..mid]$ 
            merged.push_back(a[j++]);
        }
    }
    for (int k = l; k <= r; k++) a[k] = merged[k - l];
    return cnt;
}

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    vector<int> a(n);
    for (int& v : a)
        if (scanf("%d", &v) != 1) return 0;
    printf("%lld\n", n ? sortCount(a, 0, n - 1) : 0LL);
    return 0;
}
```

The input 6 followed by 5 3 8 1 9 2 has eight inversions (5 before 3, 1 and 2; 3 before 1 and 2; 8 before 1 and 2; 9 before 2), and the program agrees:

8

Compare the structure with **Pseudocode 11.1**: split by position, handle both halves, and count the cross pairs in one pass that relies on the halves being sorted. The difference is that inversions need no order between the three steps, because nothing inside a half depends on the other half. In the LIS the values *do* depend on each other, and that is exactly what forces the order of Section 11.2.3.

11.3.2 The one-dimensional LIS with a Fenwick tree of maxima

The second warm-up is the tool that CDQ uses inside its cross step. **Listing 11.2** computes Equation (11.1) left to right. It first *compresses* the values: it sorts the distinct values and replaces each value by its rank $1, 2, \dots, m$, so that the Fenwick tree needs only $m \leq n$ cells even when values reach 10^9 . The query $p - 1$ asks for values strictly smaller than the current one; using p instead would allow equal values and compute the longest *non-decreasing* subsequence.

Listing 11.2 `lis_fenwick.cpp`: the classic LIS in $O(n \log n)$ with a Fenwick tree of prefix maxima.

```
// Warm-up: the classic strict LIS of one sequence in  $O(n \log n)$  with a
// Fenwick tree of maxima over compressed values.
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    vector<int> a(n);
    for (int& v : a)
        if (scanf("%d", &v) != 1) return 0;
    vector<int> vals(a);
    sort(vals.begin(), vals.end());
    vals.erase(unique(vals.begin(), vals.end()), vals.end());
    int m = (int)vals.size(), best = 0;
    vector<int> fen(m + 1, 0);
    for (int v : a) {
        int p = int(lower_bound(vals.begin(), vals.end(), v) - vals.begin()) + 1;
        int got = 0; // best chain with value < v
        for (int i = p - 1; i > 0; i -= i & -i) got = max(got, fen[i]);
        int here = got + 1;
        for (int i = p; i <= m; i += i & -i) fen[i] = max(fen[i], here);
        best = max(best, here);
        printf("value %3d -> chain %d\n", v, here);
    }
    printf("LIS length: %d\n", best);
    return 0;
}
```

On the sequence 3 1 4 1 5 9 2 6 it prints the chain length ending at every element:

```
value  3 -> chain 1
value  1 -> chain 1
value  4 -> chain 2
value  1 -> chain 1
value  5 -> chain 3
value  9 -> chain 4
value  2 -> chain 2
value  6 -> chain 4
LIS length: 4
```

The longest chain has length 4, for example 3, 4, 5, 9 or 1, 4, 5, 6. Note that the second 1 gets length 1, not 2: strictness is enforced by the `p - 1`.

A Fenwick tree of *maxima* has one limitation worth knowing: values may only grow. You cannot lower a cell back to a smaller value, because the maximum stored higher up would not know. In CDQ this is not a problem, because we never lower a value; we only reset cells to 0, walking the same path, after the whole cross step is finished.

11.4 Case Study: LIS2

11.4.1 The problem in our own words

The SPOJ problem LIS2 [6] gives N points (x_i, y_i) in a fixed order, with N up to 10^5 . Find the largest k for which there are positions $i_1 < i_2 < \dots < i_k$ such that both $x_{i_1} < x_{i_2} < \dots < x_{i_k}$ and $y_{i_1} < y_{i_2} < \dots < y_{i_k}$. Print that k . The input is the number N followed by N lines with two integers each. This is Dewi's notebook, with distance as x and speed as y .

11.4.2 A sample by hand

Here is our own small input: the eight points of **Figure 11.1**.

```
8
1 3
3 2
1 1
4 5
6 3
9 9
8 7
7 6
```

Work **Equation (11.2)** by hand. Point 0 is (1,3) and gets 1. Point 1, (3,2), cannot follow (1,3) because $2 < 3$ in y ; it gets 1. Point 2, (1,1), gets 1. Point 3, (4,5), can follow (1,3), (3,2) or (1,1): 2. Point 4, (6,3), can follow (3,2) or (1,1): 2. Point 5, (9,9), can follow (4,5): 3. Point 6, (8,7), can follow (4,5) or (6,3): 3. Point 7, (7,6), can follow (4,5) or (6,3): 3; it cannot follow (8,7) or (9,9), which are larger. The answer is 3, for instance $(1,1) \rightarrow (4,5) \rightarrow (9,9)$, and the program prints:

```
3
```

In the CDQ run on this input, the first cross step at the top level uses exactly the situation of **Figure 11.1(b)**: when the sweep reaches $p = (7,6)$, all four left points have a smaller x , the ones with $y < 6$ are (1,3), (3,2), (1,1) and (4,5), and the best of their final values is 2, so p receives 3.

11.4.3 Design

The design follows **Pseudocode 11.1** with three practical decisions.

1. **Compress y .** Coordinates can be large or negative, but only their order matters. We replace y by its rank among the distinct y values, $1..m$, so the Fenwick tree has $m + 1$ cells.
2. **Sort index lists, not points.** The cross step sorts *copies of the indices* of each half by x ; the arrays `px`, `py` and `dp` stay in input order. That way the recursion into the right half still sees positions in input order, which it needs.
3. **Strict inequalities.** Left points are inserted while `px[left[j]] < px[i]`, so a left point with the *same* x as i is never inserted before i is answered; and the query reads positions up to `py[i] - 1`, so equal y values are excluded as well.

11.4.4 The complete solution

Listing 11.3 is the full program. It compiles without warnings under `-std=c++17 -O2 -Wall -Wextra` and passes the stress test of Section 11.5. The part between the two `<P50>` markers is the second half of the core algorithm, the sweep of the cross step; the comment lines right after `// <P50>` say precisely what that part must do.

Listing 11.3 `LIS2.cpp`: the two-dimensional LIS in $O(n \log^2 n)$ with CDQ divide and conquer.

```
// LIS2: the longest chain of points, taken in input order, whose x AND y
// both increase strictly. CDQ divide and conquer + a Fenwick tree of maxima.
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

int n, m; // number of points; number of distinct y values
vector<int> px, py, dp; // x, compressed y (1..m), best chain ending here
vector<int> fen; // Fenwick tree: prefix maxima of dp over y

void fenUpdate(int i, int v) {
    for (; i <= m; i += i & -i) fen[i] = max(fen[i], v);
}
int fenQuery(int i) { // the largest value stored at positions 1..i
    int r = 0;
    for (; i > 0; i -= i & -i) r = max(r, fen[i]);
    return r;
}
void fenClear(int i) { // undo every write that touched position i
    for (; i <= m; i += i & -i) fen[i] = 0;
}

// Every point in [l, mid] is final; let it improve the points in [mid+1, r].
void cross(int l, int mid, int r) {
    vector<int> left, right;
    for (int i = l; i <= mid; i++) left.push_back(i);
    for (int i = mid + 1; i <= r; i++) right.push_back(i);
    auto byX = [](int a, int b) { return px[a] < px[b]; };
    sort(left.begin(), left.end(), byX);
    sort(right.begin(), right.end(), byX);
    // <P50>
    // Sweep the right points by x; first insert every left point with a smaller
    // x (value dp at its y), then extend the best chain whose y is below ours.
    // Afterwards clear only the Fenwick positions that were written.
    // >>> YOUR TURN (Public Edition): write this part yourself, following the hint above.
    <<<
    // </P50>
}
```

Listing 11.3 (Continued) LIS2.cpp: the two-dimensional LIS in $O(n \log^2 n)$ with CDQ divide and conquer.

```
void solve(int l, int r) {
    if (l >= r) return;
    int mid = (l + r) / 2;
    solve(l, mid);           // 1. the left half becomes final
    cross(l, mid, r);        // 2. it passes its influence to the right half
    solve(mid + 1, r);       // 3. only now may the right half finish
}

int main() {
    if (scanf("%d", &n) != 1) return 0;
    px.resize(n), py.resize(n), dp.assign(n, 1);
    for (int i = 0; i < n; i++)
        if (scanf("%d %d", &px[i], &py[i]) != 2) return 0;
    vector<int> ys(py);       // compress y to 1..m
    sort(ys.begin(), ys.end());
    ys.erase(unique(ys.begin(), ys.end()), ys.end());
    m = (int)ys.size();
    for (int i = 0; i < n; i++)
        py[i] = int(lower_bound(ys.begin(), ys.end(), py[i]) - ys.begin()) + 1;
    fen.assign(m + 1, 0);
    solve(0, n - 1);
    printf("%d\n", n ? *max_element(dp.begin(), dp.end()) : 0);
    return 0;
}
```

11.4.5 Line by line

Global state. `px` holds x in input order, `py` holds the compressed y , and `dp[i]` holds the best chain length known so far for a chain ending at point i . All `dp` start at 1, because a single point is a chain. The Fenwick tree `fen` has $m + 1$ cells, where cell 0 is unused.

The three Fenwick helpers. `fenUpdate(i, v)` raises every cell responsible for position i to at least v ; `fenQuery(i)` returns the maximum stored over positions $1..i$ by walking down with `i -= i & -i`; `fenClear(i)` walks exactly the path of `fenUpdate` and writes 0. The expression `i & -i` is the lowest set bit of i , the size of the block a cell covers [2].

cross(l, mid, r). The first six lines build `left` and `right`, the index lists of the two halves, and sort both by x . Then comes the sweep. The pointer `j` moves through `left` only forward: for each right point i , in increasing x , the `while` loop inserts every left point whose x is strictly smaller. Because the right points are visited in increasing x , a left point inserted for one right point is also allowed for every later one, so no point is ever inserted twice, and the whole sweep is linear apart from the Fenwick costs. The query `fenQuery(py[i] - 1)` gives the best `dp` among inserted points with a smaller y ; if none exists, it returns 0 and `dp[i]` stays at least 1. The last line undoes exactly the `j` insertions.

solve(l, r). Three calls, in the only correct order, as argued in Section 11.2.3.

main. It reads the points, builds the sorted list of distinct y values with `sort` and `unique`, and replaces each y by `lower_bound + 1`. After `solve(0, n - 1)` every `dp` is final, and the answer is the largest one. The guard `n ? ... : 0` avoids calling `max_element` on an empty vector.

11.4.6 Traps

TRAP. Four mistakes cost contestants this problem. (1) *Wrong call order*: calling CROSS after both recursive calls. It passes the sample and fails on long chains. (2) *Non-strict comparisons*: `<=` in the `while` loop or `py[i]` instead of `py[i] - 1` in the query counts equal coordinates as increasing. (3) *Clearing with fill*: resetting the whole Fenwick tree in every cross step turns

$O(n \log^2 n)$ into $O(n^2)$ and times out. (4) *Sorting the arrays themselves*: reordering `px` and `py` by x in the cross step destroys the input order that the right-half recursion depends on; sort index lists instead.

11.5 Testing It: CDQ Against the Quadratic DP

A fast program that is wrong is worth nothing, so every case study in this book ends with a *stress test*: a slow but obviously correct program, a generator of small random inputs, and a script that compares the two on hundreds of inputs. For LIS2 the obvious program is Equation (11.2) written directly, in Listing 11.4.

Listing 11.4 `brute.cpp`: the $O(n^2)$ dynamic program used as the reference.

```
// LIS2 reference: the plain  $O(n^2)$  dynamic program, for stress testing only.
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    vector<int> x(n), y(n), dp(n, 1);
    for (int i = 0; i < n; i++)
        if (scanf("%d %d", &x[i], &y[i]) != 2) return 0;
    int best = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++)
            if (x[j] < x[i] && y[j] < y[i]) dp[i] = max(dp[i], dp[j] + 1);
        best = max(best, dp[i]);
    }
    printf("%d\n", best);
    return 0;
}
```

The generator in Listing 11.5 produces up to 14 points with coordinates in a tiny range such as $[-3, 3]$. Tiny ranges are deliberate: they make equal x and equal y values common, which is where strictness bugs live. The seed comes from the command line, so every failing case can be reproduced exactly.

Listing 11.5 `gen.py`: random small inputs with many ties.

```
# Random LIS2 input: few points and a tiny coordinate range, so ties are common.
import random
import sys

rnd = random.Random(int(sys.argv[1]))
n = rnd.randint(1, 14)
k = rnd.randint(1, 6)
print(n)
for _ in range(n):
    print(rnd.randint(-k, k), rnd.randint(-k, k))
```

The driver in Listing 11.6 compiles both programs with all warnings on and compares their outputs byte for byte. The same driver, with a different problem name, is used in Chapters 12 to 15.

Listing 11.6 `stress.sh`: the stress-test driver.

```
#!/usr/bin/env bash
# Stress test: LIS2.cpp against brute.cpp on random inputs made by gen.py.
# Usage: ./stress.sh [number_of_tests] (default 500)
set -e
N=${1:-500}
g++ -std=c++17 -O2 -Wall -Wextra -o sol LIS2.cpp
g++ -std=c++17 -O2 -Wall -Wextra -o brute brute.cpp
for i in $(seq 1 "$N"); do
    python3 gen.py "$i" > in.txt
    ./sol < in.txt > out_sol.txt
    ./brute < in.txt > out_brute.txt
    if ! cmp -s out_sol.txt out_brute.txt; then
        echo "Mismatch on seed $i:"; cat in.txt; exit 1
    fi
done
echo "All $N tests passed."
```

Running `./stress.sh` prints:

```
All 500 tests passed.
```

It is instructive to break the program on purpose and watch the stress test catch it. Swap the lines `cross(1, mid, r);` and `solve(mid + 1, r);` and run again: within a handful of seeds the script stops at an input whose true answer is 3 or 4 and prints a smaller number. Change `px[left[j]] < px[i]` into `<=` and the first mismatch appears on an input where two points share an x value. This habit, breaking the solution to see whether the test notices, tells you whether your tests are strong enough.

We also compiled the program with `-fsanitize=address,undefined` and ran the sample and all random inputs; no out-of-range access or overflow was reported.

11.6 Contest-Day Tactics

Speed matters only when it is too little, so we measured. **Table 11.1** lists the largest inputs we tried and the time taken on our test machine (g++ 13, -O2, Linux x86-64).

Table 11.1 Running time of `LIS2.cpp` on the largest inputs.

Input	N	Answer	Time
random coordinates in $[-10^9, 10^9]$	100 000	107	0.15 s
the diagonal (i, i)	100 000	100 000	0.06 s

Both cases finish far below a second, so there is no need for micro-optimizations such as replacing the `vector` allocations in `cross` with static arrays. Spend the saved time on correctness. Some habits help on the day.

- **Recognize the shape.** A recurrence “value of i depends on earlier j with $a_j < a_i$ and $b_j < b_i$ ” is the signal for CDQ. So are offline problems with *updates and queries in time order*, where “earlier update affects later query” plays the role of $j < i$; there, CDQ splits the operation list by time and the cross step applies left updates to right queries [4].
- **Write the brute force first.** The quadratic DP takes three minutes to write, and it both confirms your understanding of the sample and becomes the reference for the stress test.
- **Keep the three calls on three lines with comments.** It is the single most error-prone spot, and a comment such as `// 1. left final` makes a wrong order stand out when you reread the code.

- **Know the alternatives.** A 2D LIS can also be solved with a segment tree on x holding Fenwick trees on y (online, more memory), or with a sweep over x and a Fenwick tree over *positions* if positions play the role of the third coordinate. CDQ is usually the shortest code with the smallest memory.

11.6.1 CDQ over time: updates and queries

The same skeleton solves problems that have nothing to do with subsequences. Imagine a list of operations in time order: “add a point (x, y) ” and “how many points so far lie in the rectangle $[0, x] \times [0, y]$?”. An online solution needs a two-dimensional structure. CDQ answers all queries offline with only a one-dimensional Fenwick tree. Split the operation list by time. Updates in the left half affect queries in the right half, and nothing else crosses the cut. In the cross step, sort the left updates and the right queries by x , sweep, add each update to a Fenwick tree of sums at position y , and let each query read the prefix sum up to its y . Queries inside each half are handled by the recursion. Here the three calls may come in any order, because a query never feeds its answer back into an update; the LIS is harder precisely because its “answers” (dp values) are also its “updates”. Seeing that difference quickly is part of recognizing which version of CDQ a problem needs.

11.7 Chapter Summary

- The two-dimensional LIS has three order conditions: position, x and y . One sweep plus one Fenwick tree can handle only two.
- CDQ divide and conquer splits by position. For pairs across the cut, the position condition is automatically true, so a sweep by x and a Fenwick tree of maxima over compressed y finish the job.
- The order *solve left, cross, solve right* is a correctness contract, proved by the invariant of Section 11.2.3.
- The cost satisfies Equation (11.3) and is $O(n \log^2 n)$, provided the Fenwick tree is cleaned only at the positions that were written.
- Strictness lives in two places: the $<$ in the sweep and the $- 1$ in the query.
- A stress test against the $O(n^2)$ DP, with tiny coordinate ranges, is the fastest way to catch the typical bugs.

11.8 Exercises

1. Run **Pseudocode 11.1** by hand on the points $(2,2), (1,1), (3,3), (2,4)$ and list every dp value after each call to CROSS.
2. Change `LIS2.cpp` so that it computes the longest chain in which x is non-decreasing and y strictly increasing. Which two characters change?
3. Explain in two or three sentences why the running time would become quadratic if `cross` reset the Fenwick tree with `fill(fen.begin(), fen.end(), 0)`.
4. Modify the program to also print one longest chain, as a list of input positions.
5. *Three-dimensional dominance counting.* Given n points (a_i, b_i, c_i) , count for every point how many others have all three coordinates strictly smaller. Show how CDQ reduces this to a cross step with a sweep and a Fenwick tree of *sums*, and state the complexity.

6. In **Listing 11.1**, the comparison `a[i] <= a[j]` takes the left element when the two are equal. What would the program count if it were `<`?

11.9 Solutions to the Exercises

1. Initially all `dp` are 1. `SOLVE(0,3)` splits into `[0,1]` and `[2,3]`. `SOLVE(0,1)` calls `CROSS(0,0,1)`: point `1, (1,1)`, cannot follow `(2,2)`, so nothing changes. `CROSS(0,1,3)`: sorted by x , the right points are `(2,4)` then `(3,3)`. For `(2,4)` we insert `(1,1)` only (the point `(2,2)` has equal x); the query below $y = 4$ gives 1, so `dp3 = 2`. For `(3,3)` we also insert `(2,2)`; the query below $y = 3$ gives 1, so `dp2 = 2`. Finally `SOLVE(2,3)` calls `CROSS(2,2,3)`: `(2,4)` cannot follow `(3,3)`. The final values are 1,1,2,2 and the answer is 2.
2. The sweep must insert left points with equal x too, so `px[left[j]] < px[i]` becomes `px[left[j]] <= px[i]`. That is a single character. The query `py[i] - 1` stays, because y remains strict. (The equal- x points inside one half are still handled correctly by the recursion, since position order is untouched.)
3. There is one cross step per internal node of the recursion tree, that is $n - 1$ of them, and each full reset costs $\theta(m)$ with m the number of distinct y values, up to n . The total is $\theta(n \cdot m)$, which is $\theta(n^2)$ in the worst case: 10^{10} writes for $n = 10^5$.
4. Store with every Fenwick value the index that produced it, for instance as a `pair<int,int>` (`dp`, `index`) compared by `max`. When `dp[i]` improves in the cross step, record `parent[i]` as that index. At the end start from the point with the largest `dp` and follow `parent` back, then reverse the list. The complexity is unchanged.
5. Sort the points by a (break ties so that equal a values do not count each other, for example by grouping equal points first). The sorted position now plays the role of time. CDQ splits that list; in the cross step, sort both halves by b , sweep the right half in increasing b , add $+1$ at position c_j in a Fenwick tree of sums for every left point with smaller b , and add the prefix sum below c_i to the count of i . Here the order of the three calls does not matter, because counts do not feed back into other counts. The cost is again $O(n \log^2 n)$ [4].
6. With `<`, equal elements would be taken from the right first and each would be counted as an inversion with every equal element still waiting on the left. The program would count pairs $i < j$ with $a_i \geq a_j$ instead of $a_i > a_j$.

References

- [1] D. Chen, “From ‘Cash’ to a class of divide-and-conquer algorithms” (in Chinese), China National Olympiad in Informatics Team Selection Papers, 2008.
- [2] P. M. Fenwick, “A new data structure for cumulative frequency tables,” *Software: Practice and Experience*, vol. 24, no. 3, pp. 327–336, 1994.
- [3] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.
- [4] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 2. Lulu Press, 2020.
- [5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.
- [6] Sphere Online Judge, “LIS2 - Another Longest Increasing Subsequence Problem.” [Online]. Available: <https://www.spoj.com/problems/LIS2/>

Chapter 12

Meet in the Middle: Halving the Exponential — ABCDEF

Some problems ask us to try “all combinations”, and the number of combinations is astronomical: 2^{40} subsets, or 100^6 ways to fill six slots. No trick inside the loop can save a program that must visit 10^{12} cases. *Meet in the middle* avoids visiting them at all. It splits the choices into two independent halves, lists every outcome of each half separately, and lets the two lists “shake hands” through a single linking value, usually with sorting and binary search. The work drops from the product of the two halves to roughly their sum. By the end of this chapter you will be able to spot problems that split this way, count matching pairs with `lower_bound` and `upper_bound`, estimate time and memory before writing a line, and solve the SPOJ problem ABCDEF, where 10^{12} tuples collapse to two lists of 10^6 numbers.

12.1 Where This Helps: Bu Sari’s Parcels

Bu Sari runs a small shipping counter in her village. Farmers bring parcels of fruit, and the courier truck that passes in the afternoon accepts a load only if its total weight lies inside a narrow window, say between 95 and 100 kilograms. On a busy day there are forty parcels on the floor. How many different selections of parcels would the truck accept?

Each parcel is either in the selection or not, so there are 2^{40} selections, about $1.1 \cdot 10^{12}$. A program that checks a billion selections per second, which is very optimistic, still needs almost twenty minutes. Yet the question has a structure that the brute force ignores. Put the first twenty parcels on the left side of the room and the other twenty on the right. Any selection is a left selection plus a right selection, and the only thing that connects them is the total weight: if the left part weighs x , the right part must weigh between $95 - x$ and $100 - x$. There are only $2^{20} \approx 10^6$ left selections and 10^6 right selections. List the right weights once, sort them, and for each left weight count the right weights inside the window with two binary searches. A million binary searches take a few milliseconds.

That is the whole idea of this chapter: when a big search space is a *product* of two independent parts connected only by one value, list each part separately and match them through that value. The technique is old; Horowitz and Sahni used it in 1974 to solve the knapsack problem exactly in about $2^{n/2}$ steps [1], and it still appears in national olympiads every year in new disguises [2], [3].

The shape hides in other stories too. A teacher preparing a mathematics competition wants to choose six questions, one from each of six topic pools, so that the total difficulty score is exactly 60. If each pool has 100 questions, there are 10^{12} selections. But the first three pools and the last three pools meet only through the total: if the first three questions add up to x , the last three must add up to $60 - x$. Two lists of 10^6 partial sums, one sort and a million binary searches answer the question in well under a second. A security engineer testing a cipher with a 40-bit key that is applied in two independent 20-bit stages uses the same trick to recover the key: encrypt forward from the plaintext with all 2^{20} first-stage keys, decrypt backward from the ciphertext with all 2^{20} second-stage keys, and look for values that appear in both lists. That attack is the reason why

“double encryption” is not twice as strong as single encryption, and it is literally called the *meet-in-the-middle attack*.

12.2 The Technique from Zero

12.2.1 The shape to look for

Suppose a candidate solution is a pair (p, q) , where p comes from a set P of “left choices” and q from a set Q of “right choices”. A brute force tries all $|P| \cdot |Q|$ pairs. Meet in the middle works when the condition on a pair can be written as

$$g(p) = h(q) \quad \text{or, more generally,} \quad g(p) + h(q) \in [A, B] \quad (12.1)$$

for some functions g and h that each look at one side only. The value $g(p)$ is the *linking value* of the left choice: once we know it, nothing else about p matters for the match. Equation (12.1) is the thing to search for when you read a problem. Sums, differences, products that can be moved to one side, XORs and remainders all fit.

12.2.2 The recipe

1. **Split** the variables or objects into two groups that interact only through the linking value.
2. **Enumerate** each group: compute the list $L = [g(p) : p \in P]$ and the list $R = [h(q) : q \in Q]$.
3. **Sort** one list, say R .
4. **Match**: for each value v in L , count the elements of R that fit v . For an equality, those are the elements equal to v ; in a sorted array they form one contiguous *run*, and its length is `upper_bound(v) - lower_bound(v)`.
5. **Add** the counts, in a 64-bit integer.

Pseudocode 12.1 writes the recipe for the equality case.

Pseudocode 12.1 Counting matching pairs by meet in the middle.

```
MITM-COUNT(P, Q)
1  L = [ g(p) for every p in P ]
2  R = [ h(q) for every q in Q ]
3  sort R
4  total = 0
5  for each v in L
6      total = total + (UPPER-BOUND(R, v) - LOWER-BOUND(R, v))
7  return total
```

DEFINITION. For a sorted array R and a value v , `lower_bound(v)` is the first position whose element is $\geq v$ and `upper_bound(v)` is the first position whose element is $> v$. The elements equal to v occupy exactly the positions from `lower_bound(v)` up to, but not including, `upper_bound(v)`, so their number is the difference of the two positions. In C++ `equal_range(v)` returns both positions at once [4].

Why the count is right. Every pair (p, q) is counted at most once: when the loop reaches $g(p)$ it counts q if and only if $h(q) = g(p)$. And every matching pair is counted at least once, for the same reason. If several left choices share the same linking value, each is its own element of L and each counts its partners, which is correct because they are different pairs. **Figure 12.1** shows the handshake on two tiny lists: the left value 3 appears twice and the right value 3 three times, so they contribute $2 \cdot 3 = 6$ pairs.

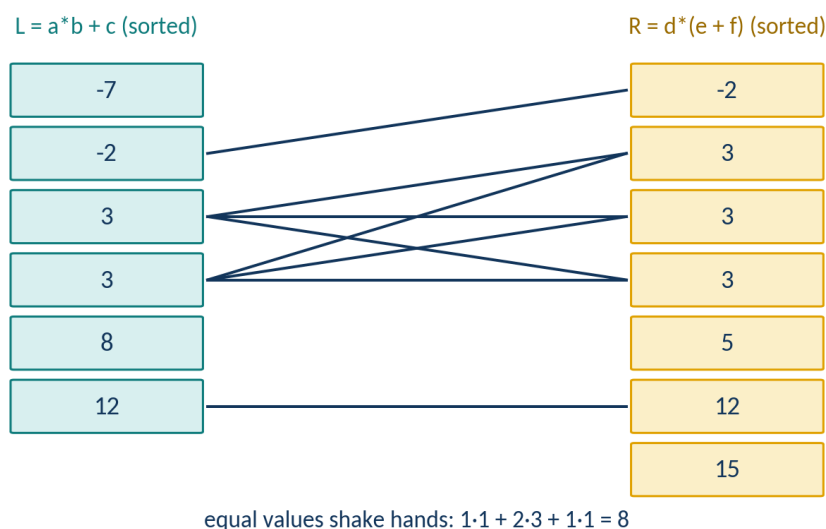


Figure 12.1 Two sorted lists and their handshakes: each line is one counted pair, and equal values form runs whose sizes multiply.

12.2.3 Time and memory, estimated before coding

Let $|L| = |R| = K$. Building the lists costs $O(K)$, sorting R costs $O(K \log K)$, and K pairs of binary searches cost $O(K \log K)$. So

$$T = O(K \log K) \quad \text{instead of} \quad O(K^2) \quad (12.2)$$

For subset sums over n items, $K = 2^{n/2}$, and Equation (12.2) turns 2^{40} into about $2^{20} \cdot 20 \approx 2 \cdot 10^7$ steps. The price is memory: both lists live in RAM at the same time. Always estimate it: 10^6 values of type `long long` take 8 MB, so two lists take 16 MB, which fits the usual 256 MB limit easily; 2^{25} values each would need 512 MB and would not. When memory is the bottleneck you can sort *both* lists and walk them with two pointers (Exercise 3), or keep only one list and regenerate the other on the fly; Schroepel and Shamir even showed how to get the memory down to about $2^{n/4}$ for subset sums while keeping the time near $2^{n/2}$ [5].

KEY INSIGHT. Meet in the middle does not make exponential problems polynomial; it takes a square root of the search space. It is the right tool exactly when $\sqrt{\text{search space}}$ is about 10^6 to 10^7 : $n \approx 40$ binary choices, or three free variables out of six with 100 values each.

12.2.4 A run by hand

Take four parcels of 3, 5, 6 and 8 kilograms and a window of [9,11] kilograms. The left half {3,5} has the subset sums 0,3,5,8; the right half {6,8} has the subset sums 0,6,8,14, already sorted. Now match:

- left sum 0 needs a right sum in [9,11]: none;
- left sum 3 needs [6,8]: the right sums 6 and 8, two subsets;
- left sum 5 needs [4,6]: the right sum 6, one subset;
- left sum 8 needs [1,3]: none.

The total is 3, and indeed the accepted selections are {3,6}, {3,8} and {5,6}. We looked at four left sums and did two binary searches for each, instead of listing all sixteen subsets. With forty parcels the same work is a million left sums instead of a trillion subsets, and that difference is the whole point of the technique.

A small detail in this run deserves attention: the left sum 3 found *two* partners. Meet in the middle never asks “is there a partner?” when the problem counts; it asks “how many partners?”, and the answer comes from the width of an interval in a sorted list, not from a yes-or-no lookup.

12.3 Warm-Up Problems

12.3.1 Counting pairs with a given sum

The smallest meet in the middle has one variable on each side: given two lists a and b and a target T , count pairs (i, j) with $a_i + b_j = T$. The linking value of a_i is $T - a_i$, the partner it needs.

Listing 12.1 sorts b and counts the run of $T - a_i$ for each i .

Listing 12.1 `two_sum_count.cpp`: counting pairs with a given sum by sorting and two binary searches.

```
// Warm-up: how many pairs (i, j) satisfy a[i] + b[j] = T?
// Sort b once; each a[i] then asks for the run of values equal to T - a[i].
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

int main() {
    int n, m;
    long long T;
    if (scanf("%d %d %lld", &n, &m, &T) != 3) return 0;
    vector<long long> a(n), b(m);
    for (auto& v : a)
        if (scanf("%lld", &v) != 1) return 0;
    for (auto& v : b)
        if (scanf("%lld", &v) != 1) return 0;
    sort(b.begin(), b.end());
    long long pairs = 0;
    for (long long v : a) {
        auto lo = lower_bound(b.begin(), b.end(), T - v);
        auto hi = upper_bound(b.begin(), b.end(), T - v);
        pairs += hi - lo;
    }
    printf("%lld\n", pairs);
    return 0;
}
```

With $a = [1, 4, 6, 9]$, $b = [4, 6, 6, 9, 1]$ and $T = 10$ the pairs are $1 + 9$, $4 + 6$ (twice, because b contains 6 twice), $6 + 4$ and $9 + 1$:

```
4 5 10
1 4 6 9
4 6 6 9 1
```

5

12.3.2 Subset sums in a window, $n \leq 40$

Bu Sari’s question from Section 12.1 is the classic SPOJ problem SUBSUMS [6]: count the subsets of $n \leq 34$ numbers whose sum lies in $[A, B]$. **Listing 12.2** solves it for up to 40 numbers. The helper `sums` lists all subset sums of one half by doubling a list: start from the empty subset, and for each new number append, for every sum already present, that sum plus the number. For the window condition, the partner y of a left sum x must satisfy $A - x \leq y \leq B - x$, and the number

of such y in the sorted right list is `upper_bound(B - x) - lower_bound(A - x)`, the second form of Equation (12.1).

Listing 12.2 `subset_mitm.cpp`: counting subsets with a sum in $[A, B]$ by meet in the middle.

```
// Warm-up: count the subsets of n <= 40 numbers whose sum lies in [A, B].
// Split into two halves of at most 20, list both halves' subset sums, sort one.
#include <algorithm>
#include <cstdint>
#include <vector>
using namespace std;

vector<long long> sums(const vector<long long>& part) {
    vector<long long> out(1, 0);           // the empty subset
    for (long long v : part) {
        size_t k = out.size();
        for (size_t i = 0; i < k; i++) out.push_back(out[i] + v);
    }
    return out;
}

int main() {
    int n;
    long long A, B;
    if (scanf("%d %lld %lld", &n, &A, &B) != 3) return 0;
    vector<long long> a(n);
    for (auto& v : a)
        if (scanf("%lld", &v) != 1) return 0;
    vector<long long> left(a.begin(), a.begin() + n / 2);
    vector<long long> right(a.begin() + n / 2, a.end());
    vector<long long> X = sums(left), Y = sums(right);
    sort(Y.begin(), Y.end());
    long long cnt = 0;
    for (long long x : X)                // need A - x <= y <= B - x
        cnt += upper_bound(Y.begin(), Y.end(), B - x)
            - lower_bound(Y.begin(), Y.end(), A - x);
    printf("%lld\n", cnt);
    return 0;
}
```

For the five numbers 2, −1, 4, 3, 5 and the window [3, 7] it prints:

```
5 3 7
2 -1 4 3 5
```

14

You can check the 14 by hand in a few minutes, or better, by a three-line brute force over all $2^5 = 32$ subsets. The empty subset counts too; its sum 0 lies outside this window, so it is not among the 14.

12.4 Case Study: ABCDEF

12.4.1 The problem in our own words

The SPOJ problem ABCDEF [7] gives a set S of n distinct integers, $1 \leq n \leq 100$, each between −30 000 and 30 000. Count the six-tuples (a, b, c, d, e, f) of elements of S (repetitions allowed) that satisfy

$$\frac{a \cdot b + c}{d} - e = f, \quad d \neq 0 \quad (12.3)$$

The input is n followed by the elements; the output is the count.

12.4.2 From one equation to two sides

A brute force over all six variables costs $n^6 = 10^{12}$ steps. The first move is algebra. Add e to both sides of Equation (12.3) and multiply by d , which is allowed because $d \neq 0$:

$$a \cdot b + c = d \cdot (e + f), \quad d \neq 0 \quad (12.4)$$

Now Equation (12.4) has exactly the shape of Equation (12.1): the left side depends only on (a, b, c) and the right side only on (d, e, f) . Each side has $n^3 = 10^6$ combinations. List $L =$ every $a \cdot b + c$, list $R =$ every $d \cdot (e + f)$ with $d \neq 0$, sort R , and count handshakes with **Pseudocode 12.1**. The work falls from 10^{12} to about $10^6 \cdot 20 = 2 \cdot 10^7$ comparisons.

Why this split and not another? Try moving e to the left: $a \cdot b + c - d \cdot e = d \cdot f$. Now d appears on both sides, so the sides are no longer independent. The split must put every variable on exactly one side, and Equation (12.4) is the balanced one, with three variables each.

12.4.3 Where to enforce $d \neq 0$

The condition $d \neq 0$ belongs to the right side. We enforce it *where the value is born*: the loop over d simply skips zero, so no illegal value ever enters R . The alternative, putting every product in R and subtracting the illegal matches at the end, is longer and easy to get wrong. Note that $d = 0$ would produce the value 0 for every (e, f) ; if 0 appears in L , which happens whenever $0 \in S$ or some $a \cdot b = -c$, those fake matches would be counted in large numbers.

12.4.4 The samples

The problem statement comes with four samples, and they are worth running by hand. For $S = \{1\}$ the only tuple is all ones: $1 \cdot 1 + 1 = 2 = 1 \cdot (1 + 1)$, so the count is 1. For $S = \{2, 3\}$ the answer is 4, for $S = \{-1, 1\}$ it is 24, and for $S = \{5, 7, 10\}$ it is 10. Each input is two lines, n and then the set, for example 3 followed by 5 7 10. Our program, run on the four inputs one after the other, prints:

```
1
4
24
10
```

It is worth checking one of the samples by hand, because it shows why repeated values must stay in the lists. Take $S = \{-1, 1\}$. The products $a \cdot b$ are 1, -1, -1, 1, and adding $c \in \{-1, 1\}$ gives the eight left values 0, 2, -2, 0, -2, 0, 0, 2: the value 0 four times, 2 twice and -2 twice. On the right, $e + f$ takes the values -2, 0, 0, 2; multiplying by $d = -1$ gives 2, 0, 0, -2 and by $d = 1$ gives -2, 0, 0, 2, so the right list also holds 0 four times, 2 twice and -2 twice. The handshakes are $4 \cdot 4 + 2 \cdot 2 + 2 \cdot 2 = 24$, the expected answer. Had we removed duplicates from either list, we would have counted 3.

12.4.5 The complete solution

Listing 12.3 is the full program. The first half of the core, building both lists, is outside the markers; the handshake, sorting and counting, is the part between `// <P50>` and `// </P50>`.

Listing 12.3 `ABCDEF.cpp`: counting the tuples of Equation (12.4) by meet in the middle.

```
// ABCDEF: count the six-tuples from the set S with (a*b + c) / d - e = f and
// d != 0, i.e. a*b + c = d*(e + f). Meet in the middle: n^3 values per side.
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    vector<long long> s(n);
    for (auto& v : s)
        if (scanf("%lld", &v) != 1) return 0;
    vector<long long> L, R;
    L.reserve((size_t)n * n * n);
    R.reserve((size_t)n * n * n);
    for (long long a : s)                // left side: a*b + c
        for (long long b : s)
            for (long long c : s) L.push_back(a * b + c);
    for (long long d : s) {               // right side: d*(e + f)
        if (d == 0) continue;            // d = 0 is illegal: never born
        for (long long e : s)
            for (long long f : s) R.push_back(d * (e + f));
    }
    // <P50>
    // Sort R; each Left value v matches as many right values as are equal to
    // v, which is the width of v's run in R: upper_bound - lower_bound.
    // >>> YOUR TURN (Public Edition): write this part yourself, following the hint above.
    <<<
    // </P50>
    printf("%lld\n", answer);
    return 0;
}
```

12.4.6 Line by line

Reading. The elements are read into `vector<long long> s`. Using 64-bit numbers from the start means that every product below is computed in 64 bits without casts.

The two `reserve` calls allocate room for n^3 elements once, so `push_back` never reallocates. For $n = 100$ that is 10^6 elements, 8 MB per list.

The left loops produce every $a \cdot b + c$. There are exactly n^3 values, including repeats; repeats must stay, because different tuples are different solutions.

The right loops produce every $d \cdot (e + f)$ with $d \neq 0$: at most n^3 values, fewer if $0 \in S$.

The handshake. `sort(R.begin(), R.end())` puts equal values together. For each left value `v`, `equal_range` returns the pair `(lower_bound, upper_bound)` of the run of `v` in `R`, and the difference of the two iterators is the size of the run. That is the number of right triples (d, e, f) that complete the left triple into a solution. The running `answer` is a `long long`.

Why not three loops and a lookup? A tempting shortcut is to loop over (a, b, c) and, for each left value, loop over d and check whether $v/d - e$ is in the set for some e . That is $n^4 = 10^8$ steps with divisions and set lookups: probably too slow, and full of divisibility cases. The clean split of Equation (12.4) keeps both sides symmetric and makes every step a plain integer operation.

A budget written before coding. For $n = 100$: building `L` and `R` takes $2 \cdot 10^6$ multiplications; sorting `R` takes about $10^6 \cdot 20 = 2 \cdot 10^7$ comparisons; the 10^6 calls of `equal_range` take about $2 \cdot 10^6 \cdot 20 = 4 \cdot 10^7$ comparisons. In total roughly $6 \cdot 10^7$ simple operations, well under a second. The measured time in Section 12.6 is 0.14 seconds, which matches the estimate. Making such an

estimate a habit means you rarely submit a solution that is too slow, and you never waste time optimizing one that is already fast enough.

12.4.7 Numbers that save or bury you

Check the ranges before choosing types; **Table 12.1** does it for this problem.

Table 12.1 The ranges in ABCDEF: the values fit in 32 bits, the count does not.

Quantity	Largest absolute value	Fits in <code>int</code> ?
$a \cdot b + c$	$30\,000^2 + 30\,000 = 900\,030\,000$	yes
$d \cdot (e + f)$	$30\,000 \cdot 60\,000 = 1.8 \cdot 10^9$	yes, barely
the answer	up to $n^6 = 10^{12}$	no
memory of L and R	$2 \cdot 10^6 \cdot 8 \text{ bytes} = 16 \text{ MB}$	—

The values would fit in `int`, but only with about 16% to spare on the right side, and one careless change (say a problem variant with $e + f + g$) would overflow silently. The count certainly needs 64 bits: with $S = \{-50, \dots, 49\}$ our program reports 349 653 095 tuples, and larger sets can give more than $2^{31} \approx 2.1 \cdot 10^9$.

TRAP. (1) An `int` accumulator overflows on large answers. (2) Adding the $d = 0$ products to R counts impossible tuples. (3) Removing repeated values from L or R with `unique` loses solutions: every triple is a separate way. (4) Reading the equation as integer division, and trying to check that d divides $a \cdot b + c$ exactly, is unnecessary: Equation (12.4) is equivalent to Equation (12.3) for $d \neq 0$, and it is what the judge counts.

12.5 Testing It: Meet in the Middle Against n^6

The reference for ABCDEF is the equation itself, checked for every tuple, in **Listing 12.4**. With $n \leq 6$ it performs at most $6^6 = 46\,656$ checks.

Listing 12.4 `brute.cpp`: the n^6 reference.

```
// ABCDEF reference: try all n^6 tuples (only for tiny n).
#include <cstdio>
#include <vector>
using namespace std;

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    vector<long long> s(n);
    for (auto& v : s)
        if (scanf("%lld", &v) != 1) return 0;
    long long cnt = 0;
    for (long long a : s) for (long long b : s) for (long long c : s)
        for (long long d : s) for (long long e : s) for (long long f : s)
            if (d != 0 && a * b + c == d * (e + f)) cnt++;
    printf("%lld\n", cnt);
    return 0;
}
```

The generator in **Listing 12.5** draws up to six distinct values from a small range around zero, so zero is often in the set (testing the $d \neq 0$ rule) and many tuples match (testing the run counting). Occasionally it uses the full range $\pm 30\,000$ to test large products.

Listing 12.5 `gen.py`: small random sets for ABCDEF.

```
# Random ABCDEF input: up to 6 distinct values from a small range (0 included
# often), so that many tuples match.
import random
import sys

rnd = random.Random(int(sys.argv[1]))
k = rnd.choice([3, 5, 10, 30000])
n = rnd.randint(1, min(6, 2 * k + 1))
vals = rnd.sample(range(-k, k + 1), n)
print(n)
print(*vals)
```

The driver is the same `stress.sh` as in Chapter 11 with `ABCDEF.cpp` in place of `LIS2.cpp`. It reports:

```
All 500 tests passed.
```

12.6 Contest-Day Tactics

Table 12.2 shows the program on the largest inputs, measured on our test machine (g++ 13, -O2).

Table 12.2 Running time of `ABCDEF.cpp` at the maximum size.

Input ($n=100$)	Answer	Time
100 random distinct values in $[-30\,000, 30\,000]$	2 700	0.14 s
the values $-50, -49, \dots, 49$	349 653 095	0.12 s

A few lessons for the contest hall:

- **Read the limits as a hint.** $n \leq 40$ for subsets, or six variables over 100 values, practically announces meet in the middle: the square root of the brute force is about 10^6 .
- **Do the algebra on paper first.** The split in Equation (12.4) took two lines of algebra and saved 10^{12} steps. Try to move every variable to the side where it lives alone.
- **Prefer sorting to map.** A `map<long long, int>` of counts would also work, but it is several times slower and uses much more memory for 10^6 keys. `unordered_map` can be attacked by special inputs on some judges. A sorted `vector` with binary search is fast, small and predictable.
- **Two pointers are even faster.** Sort both L and R , then walk them together: when the current values are equal, count the run in each and add the product of the run lengths. This replaces 10^6 binary searches by one linear pass (Exercise 3).
- **Estimate memory aloud.** “Two lists of a million 8-byte numbers, 16 MB” takes five seconds to say and prevents a memory-limit verdict.

12.6.1 Relatives of meet in the middle

The same handshake appears in several well-known forms, and recognizing them saves time in a contest.

- **XOR instead of sum.** Count subsets whose XOR equals a target K : the partner of a left XOR x is the right XOR $x \oplus K$. Sorting and binary search work unchanged, because equality is all that is needed.
- **Bidirectional search.** In a puzzle whose states form a huge graph (sliding tiles, rotating rings, a scrambled cube), a breadth-first search from the start to depth k and another from the goal to depth k meet in the middle; states found by both searches link the two halves. If each state has b successors, the work falls from about b^{2k} to about $2b^k$.

- **Baby-step giant-step.** To solve $a^x \equiv b \pmod{p}$, write $x = i \cdot m - j$ with $m \approx \sqrt{p}$. Store all $b \cdot a^j$ for $0 \leq j < m$ (baby steps), then try a^{im} for $i = 1, 2, \dots$ (giant steps) until one matches. The time is about $\sqrt{p}$ instead of p [3].
- **Four lists.** “Choose one element from each of four lists so that the sum is zero” splits into two pairs of lists, exactly as in Exercise 5.

In each case, ask the same three questions: what are the two halves, what is the linking value, and how large is each list? The answers tell you immediately whether the method fits in time and memory.

12.6.2 Common mistakes

Beyond the traps of ABCDEF, three general mistakes deserve a name. The first is splitting *unevenly*: halves of 15 and 25 items give lists of $3 \cdot 10^4$ and $3 \cdot 10^7$ values, and the larger list dominates both time and memory, so always split as evenly as the structure allows. The second is *sorting the wrong list*: sort the list you search in, and iterate over the other one; if one list is much smaller, it is usually better to sort the larger one once and iterate over the smaller one. The third is using a *hash map with unlucky keys*: on some judges crafted inputs make `unordered_map` degrade to linear time per lookup, and a sorted vector never has that problem.

12.7 Chapter Summary

- Meet in the middle splits a product-shaped search space into two halves connected by one linking value, as in Equation (12.1).
- Each half is enumerated into a list; one list is sorted; matching values are counted by `upper_bound - lower_bound` or `equal_range`.
- The cost drops from K^2 to $O(K \log K)$ (Equation (12.2)), at the price of storing both lists.
- ABCDEF becomes Equation (12.4): $a \cdot b + c = d \cdot (e + f)$, two lists of n^3 values, with $d \neq 0$ enforced where the right values are created.
- Repeated values must stay in the lists, and the count must be 64-bit.

Meet in the middle is a reminder that a big number in the limits is not always a wall. When the search space is a product, look for the seam along which it can be cut, and let the two halves meet there.

12.8 Exercises

1. For $S = \{2, 3\}$, list the left values $a \cdot b + c$ and the right values $d \cdot (e + f)$ and verify by hand that the answer is 4.
2. Explain why moving e to the left side of Equation (12.4), as in $a \cdot b + c - d \cdot e = d \cdot f$, destroys the meet in the middle.
3. Rewrite the counting part of `ABCDEF.cpp` so that it sorts both lists and counts matches with two pointers. What is the new complexity of the matching phase?
4. Given $n \leq 40$ numbers and a target S , decide whether some subset sums to *exactly* S and print one such subset.
5. *4SUM counting*. Given four lists A, B, C, D of length $n \leq 4000$, count the quadruples with $a + b + c + d = 0$. Which split do you use, and how much memory does it need?

6. The problem asks for $n \leq 100$. Estimate the time and memory if the limit were $n \leq 300$. Would the approach still work?
7. Given $n \leq 36$ items, each with a weight and a value, find the largest total value with total weight at most W . How does meet in the middle apply when the linking condition is an inequality and we want a maximum, not a count?
8. Explain why the answer to ABCDEF for $S = \{0\}$ is 0, and compute the answer for $S = \{0,1\}$ by reasoning about the two lists.

12.9 Solutions to the Exercises

1. Left values, for (a, b, c) in order 222,223,232,233,322,323,332,333: 6,7,8,9,8,9,11,12. Right values for (d, e, f) : with $d = 2$ the sums $e + f$ are 4,5,5,6, giving 8,10,10,12; with $d = 3$ they give 12,15,15,18. Matches: left 8 appears twice and right 8 once (2 pairs); left 12 appears once and right 12 twice (2 pairs); no other value is shared. Total $2 + 2 = 4$.
2. After the move, d appears on both sides. A left value can no longer be computed without knowing the right triple, so the two lists are not independent and cannot be enumerated separately. Every variable must live on exactly one side.
3. Sort L and R . Keep indices i and j . If $L[i] < R[j]$ advance i ; if greater, advance j ; if equal to a value v , count the run length p of v in L and q in R , add $p * q$ (as `long long`), and skip both runs. Matching is now $O(n^3)$ after the two sorts, so the total is dominated by sorting, $O(n^3 \log n)$.
4. Split into halves of size $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor$. Store the right sums together with their subset masks as pairs, and sort by sum. For each left sum x with mask m_1 , binary search for $S - x$; if found with mask m_2 , print the elements selected by m_1 and m_2 . Time $O(2^{n/2} \cdot n)$.
5. Split into (a, b) and (c, d) . Build $L = [a + b]$ and $R = [-(c + d)]$, each of size $n^2 = 1.6 \cdot 10^7$, sort R , and count handshakes. Memory: two lists of $1.6 \cdot 10^7$ values; with 32-bit `int` values, 64 MB each, 128 MB in total, still inside 256 MB. With `long long` it would double, so check the value range first (sums of two 32-bit values may need 64 bits).
6. $n^3 = 2.7 \cdot 10^7$ values per list: 216 MB each as `long long`, 432 MB together, too much for 256 MB. The values fit in 32 bits, since $|a \cdot b + c| \leq 9 \cdot 10^8$ and $|d \cdot (e + f)| \leq 1.8 \cdot 10^9 < 2^{31}$, so `int` lists would need 108 MB each, 216 MB in total, which is risky. Sorting $2.7 \cdot 10^7$ values takes about two seconds. The approach is borderline and would need 32-bit lists and two pointers to be safe.
7. List all subsets of each half as pairs (weight, value). Sort the right pairs by weight and, walking in increasing weight, replace each value by the best value seen so far, so that the list answers “the best value with weight at most w ” by binary search. Remove dominated pairs if you like. For each left pair (w_1, v_1) with $w_1 \leq W$, binary search the largest right weight $\leq W - w_1$ and take v_1 plus the prefix-best value there. Keep the maximum. Time $O(2^{n/2} \cdot n)$.
8. With $S = \{0\}$ the only possible d is 0, which is forbidden, so the right list is empty and there are no solutions. With $S = \{0,1\}$: the left values $a \cdot b + c$ over 8 triples are 0 (from $a \cdot b = 0$ and $c = 0$: 3 triples), 1 (from $a \cdot b = 0$, $c = 1$: 3 triples, and from $a \cdot b = 1$, $c = 0$: 1 triple, 4 in all) and 2 (1 triple). On the right $d = 1$, so the values are $e + f$: 0 once, 1 twice, 2 once. The handshakes are $3 \cdot 1 + 4 \cdot 2 + 1 \cdot 1 = 12$.

References

- [1] E. Horowitz and S. Sahni, "Computing partitions with applications to the knapsack problem," *Journal of the ACM*, vol. 21, no. 2, pp. 277–292, 1974.
- [2] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Lulu Press, 2020.
- [3] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.
- [4] cppreference.com, "std::equal_range." [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/equal_range
- [5] R. Schroeppe and A. Shamir, "A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-complete problems," *SIAM Journal on Computing*, vol. 10, no. 3, pp. 456–464, 1981.
- [6] Sphere Online Judge, "SUBSUMS - Subset Sums." [Online]. Available: <https://www.spoj.com/problems/SUBSUMS/>
- [7] Sphere Online Judge, "ABCDEF - ABCDEF." [Online]. Available: <https://www.spoj.com/problems/ABCDEF/>

Chapter 13

Suffix Arrays and LCP: Counting Distinct Substrings — DISUBSTR

Questions about the substrings of a text come up in every olympiad: how many different pieces does a word contain, where does a pattern occur, what is the longest piece that appears twice? Each of these can be answered slowly by looking at all $O(n^2)$ substrings. This chapter builds one structure that answers all of them quickly: the *suffix array*, the list of all suffixes of the text in sorted order, together with the *LCP array*, which records how much each suffix shares with its neighbor in that order. You will learn to build the suffix array by *prefix doubling* in $O(n \log^2 n)$, to compute the LCP array in $O(n)$ with Kasai's algorithm, and to prove the one-line formula that solves the SPOJ problem DISUBSTR: the number of distinct substrings equals $n(n + 1)/2$ minus the sum of the LCP values.

13.1 Where This Helps: Words in a Batik Pattern

A batik workshop describes each of its patterns as a long string of motif codes: A for a kawung circle, B for a parang slash, C for a truntum flower, and so on. The designers want to know how *rich* a pattern is. One simple measure is the number of different stretches of motifs it contains. The pattern ABABA looks long, but it contains only nine different stretches: A, B, AB, BA, ABA, BAB, ABAB, BABA and ABABA. The pattern CCCCC contains only five.

The same count measures the complexity of DNA fragments in biology, helps compressors decide whether a text is repetitive, and appears in plagiarism detection, where long shared substrings between two essays are suspicious. In all these cases the texts are long, and listing all substrings is hopeless: a text of length n has $n(n + 1)/2$ substrings (counted with repeats), and comparing them costs even more. We want a method that looks at the text only a few times.

The key is a simple observation that we will use throughout the chapter.

KEY INSIGHT. Every substring is a *prefix of a suffix*. The substring that starts at position i and has length ℓ is the first ℓ characters of the suffix that starts at i . So to understand all substrings, it is enough to understand all n suffixes and their prefixes.

Why not simply generate all substrings and put them into a set, as the brute force of Section 13.5 does? Because the cost grows very fast. A string of 1000 characters has 500 500 substrings with an average length of about 333 characters, so the set would hold on the order of 10^8 characters and every insertion would compare long strings. For 20 such strings that is already slow, and for 10^5 characters it is impossible: $5 \cdot 10^9$ substrings. The suffix array stores only n integers, and the LCP array another n . All the information about the $n(n + 1)/2$ substrings is packed into those $2n$ numbers, and Equation (13.4) reads the answer out of them with one pass. This compression, a quadratic amount of structure described by a linear amount of data, is what makes suffix arrays so powerful.

13.2 The Technique from Zero

13.2.1 Suffix array and LCP array

Number the characters of a string s from 0 to $n - 1$, and write $s[i..]$ for the suffix that starts at position i . The *suffix array* sa is the list of starting positions sorted so that the suffixes are in lexicographic (dictionary) order:

$$s[sa[0]..] < s[sa[1]..] < \dots < s[sa[n-1]..] \quad (13.1)$$

The *LCP array* stores, for every $k \geq 1$, the length of the longest common prefix of two neighbors in that order:

$$lcp[k] = \text{LCP}(s[sa[k-1]..], s[sa[k]..]), \quad lcp[0] = 0 \quad (13.2)$$

For $s = \text{ABABA}$ the five suffixes in sorted order are A, ABA, ABABA, BA and BABA, so $sa = [4, 2, 0, 3, 1]$ and $lcp = [0, 1, 3, 0, 2]$. **Figure 13.1** draws this table, and the shading already hints at the counting formula of Section 13.2.4. The suffix array was introduced by Manber and Myers [1] as a compact alternative to the suffix tree; it uses only a few integer arrays of length n .

k	sa[k]	lcp	suffix	
0	4	0	A	
1	2	1	A B A	
2	0	3	A B A B A	
3	3	0	B A	
4	1	2	B A B A	

already counted
 new substrings ($15 - 6 = 9$)

Figure 13.1 The sorted suffixes of ABABA. The first $lcp[k]$ characters of each suffix (grey) repeat its upper neighbor; only the remaining prefixes (teal) are new substrings, $15 - 6 = 9$ of them.

13.2.2 Building the suffix array by prefix doubling

Sorting the suffixes with ordinary string comparison works, but each comparison can cost $O(n)$ characters, so the sort costs $O(n^2 \log n)$. Prefix doubling avoids long comparisons by sorting the suffixes by their first 1, 2, 4, 8, ... characters, each round using the ranks of the previous round.

Let $rk_g[i]$ be the rank of the first g characters of suffix i among the first g characters of all suffixes: equal prefixes share a rank, and smaller prefixes get smaller ranks. The first $2g$ characters of suffix i are its first g characters followed by the first g characters of suffix $i + g$. Therefore comparing the first $2g$ characters of two suffixes is the same as comparing the *pairs*

$$(rk_g[i], rk_g[i + g]) \quad (13.3)$$

where $rk_g[i + g] = -1$ if $i + g \geq n$ (a suffix that ends early is smaller, just as “AB” comes before “ABA” in a dictionary). Sort the positions by these pairs, give equal pairs equal ranks, and we have rk_{2g} . Start with $rk_1[i] = s[i]$ (the character code) and double g until all ranks are different, which happens after at most $\lceil \log_2 n \rceil$ rounds. **Pseudocode 13.1** writes it down.

Pseudocode 13.1 Building the suffix array by prefix doubling.

```

SUFFIX-ARRAY(s, n)
1  for i = 0 to n - 1
2      sa[i] = i; rk[i] = code of s[i]
3  g = 1
4  while g < n
5      sort sa by the pair (rk[i], rk[i+g] or -1 if i+g >= n)
6      tmp[sa[0]] = 0
7      for k = 1 to n - 1
8          tmp[sa[k]] = tmp[sa[k-1]] + (pair of sa[k-1] < pair of sa[k] ? 1 : 0)
9      rk = tmp
10     if rk[sa[n-1]] == n - 1: stop      // all ranks distinct
11     g = 2g
12 return sa
    
```

A run by hand on ABABA. With $g = 1$ the pairs of positions 0 to 4 are (A,B), (B,A), (A,B), (B,A), (A,−1). Sorted: position 4, then 0 and 2 (equal pairs), then 1 and 3 (equal pairs). The new ranks are $rk_2 = [1,2,1,2,0]$ for positions 0 to 4: not yet all distinct. With $g = 2$ the pairs are (1,1), (2,2), (1,0), (2,−1), (0,−1). Sorted: position 4 (0,−1), position 2 (1,0), position 0 (1,1), position 3 (2,−1), position 1 (2,2). All pairs differ, the ranks become 0..4, and $sa = [4,2,0,3,1]$, as in **Figure 13.1**.

Cost. Each round is one comparison sort, $O(n \log n)$, with $O(1)$ comparisons of integer pairs; there are at most $\lceil \log_2 n \rceil$ rounds. The total is $O(n \log^2 n)$. Replacing the comparison sort by two passes of counting sort gives $O(n \log n)$, and linear-time constructions exist as well [2], but for DISUBSTR, where $n \leq 1000$, the simple version runs in microseconds.

Why doubling is correct. The argument is an induction on the rounds. After the first round, the ranks order the suffixes correctly by their first character, with equal characters sharing a rank. Suppose that after some round the ranks order the suffixes correctly by their first g characters. Two suffixes i and j compare on their first $2g$ characters exactly as their first halves compare, and, if those are equal, as their second halves compare. The first halves are ranked by $rk_g[i]$ and $rk_g[j]$; the second halves are the first g characters of suffixes $i + g$ and $j + g$, ranked by $rk_g[i + g]$ and $rk_g[j + g]$, or missing (rank −1) when a suffix is too short. So sorting by the pair **Equation** (13.3) orders the suffixes correctly by their first $2g$ characters, and giving equal pairs equal ranks keeps the invariant. Once $2g \geq n$, the first $2g$ characters of every suffix are the whole suffix, and all ranks are distinct because different suffixes have different lengths.

Another run, on CCCCC. In the first round every pair is (C, C) except the last suffix, whose pair is (C, −1); so suffix 4 gets rank 0 and all others rank 1. In the second round the pairs are (1,1) for suffixes 0 and 1, (1,0) for suffix 2, (1,−1) for suffix 3 and (0,−1) for suffix 4; sorted, that gives suffix 4, then 3, then 2, then 0 and 1 together. In the third round, with $g = 4$, suffix 1 has the pair (3,−1) and suffix 0 the pair (3,0), so suffix 1 comes first and every rank is now distinct: $sa = [4,3,2,1,0]$, the suffixes C, CC, CCC, CCCC, CCCCC. The LCP array is [0,1,2,3,4], and **Equation** (13.4) gives $15 - 10 = 5$. A string of equal characters is the worst case for doubling, because the ranks become distinct only in the last round.

13.2.3 The LCP array in linear time: Kasai's algorithm

Computing each $lcp[k]$ by comparing characters from scratch costs $O(n)$ per neighbor pair, $O(n^2)$ in total. Kasai, Lee, Arimura, Arikawa and Park found a way to reuse work [3]. Visit the suffixes in *text order* $i = 0, 1, \dots, n - 1$, not in sorted order. The key fact is:

LEMMA (Kasai). Let h be the LCP of suffix i with its predecessor in the sorted order. Then the LCP of suffix $i + 1$ with its predecessor is at least $h - 1$.

Proof sketch. Let suffix j be the predecessor of suffix i , sharing $h \geq 1$ characters. Remove the first character from both: suffixes $j + 1$ and $i + 1$ share $h - 1$ characters, and suffix $j + 1$ is still smaller than suffix $i + 1$, because the first characters were equal. So some suffix smaller than suffix $i + 1$ shares $h - 1$ characters with it. The immediate predecessor of suffix $i + 1$ lies between suffix $j + 1$ and suffix $i + 1$ in sorted order, and any suffix between two strings that share $h - 1$ characters also shares at least those $h - 1$ characters with both. ▀

So when we move from i to $i + 1$ we can start comparing at $h - 1$ instead of 0. The value h increases by one per successful character comparison and decreases by at most one per step, and it never exceeds n ; therefore the total number of comparisons is at most $2n$, and the whole LCP array costs $O(n)$.

13.2.4 From the LCP array to the number of distinct substrings

Go down the sorted suffixes. Suffix $sa[k]$ has $n - sa[k]$ prefixes, one for each length. Its first $lcp[k]$ prefixes are also prefixes of the previous suffix, so they have been counted already. Its longer prefixes are new: if one of them had appeared earlier in the sorted order, as a prefix of some suffix $s[sa[k']..]$ with $k' < k$, then every suffix between them in sorted order, including $sa[k - 1]$, would share that prefix, contradicting the definition of $lcp[k]$. So suffix $sa[k]$ contributes exactly $n - sa[k] - lcp[k]$ new substrings, and summing over k gives

$$\begin{aligned} \# \text{ distinct substrings} &= \sum_{k=0}^{n-1} (n - sa[k] - lcp[k]) \\ &= \frac{n(n+1)}{2} - \sum_{k=1}^{n-1} lcp[k] \end{aligned} \quad (13.4)$$

because the values $n - sa[k]$ are the numbers $1, 2, \dots, n$ in some order. For ABABA, Equation (13.4) gives $15 - (1 + 3 + 0 + 2) = 9$, matching Section 13.1; for CCCCC it gives $15 - (1 + 2 + 3 + 4) = 5$.

13.2.5 Kasai by hand on ABABA

It is worth watching the variable h move once. The suffix array is $sa = [4, 2, 0, 3, 1]$, so the inverse array is $pos[0] = 2, pos[1] = 4, pos[2] = 1, pos[3] = 3, pos[4] = 0$.

- $i = 0$: suffix ABABA sits at place 2; its predecessor is suffix 2, ABA. Starting from $h = 0$ we match A, B, A and stop at the end of ABA: $h = 3$, so $lcp[2] = 3$. Then h drops to 2.
- $i = 1$: suffix BABA at place 4; predecessor suffix 3, BA. We start at $h = 2$, which is already the end of BA, so no comparison succeeds: $lcp[4] = 2$. h drops to 1.
- $i = 2$: suffix ABA at place 1; predecessor suffix 4, A. We start at $h = 1$, the end of A: $lcp[1] = 1$. h drops to 0.
- $i = 3$: suffix BA at place 3; predecessor suffix 0, ABABA. The first characters differ: $lcp[3] = 0$.
- $i = 4$: suffix A is the smallest; there is no predecessor and h is reset to 0.

Only four character comparisons succeeded in total, and in two steps the lemma let us skip comparisons entirely. The LCP array is $[0, 1, 3, 0, 2]$, as in **Figure 13.1**.

13.3 Warm-Up Problems

13.3.1 The slow suffix array, as a table

Before writing the fast version, write the obvious one and look at its output. **Listing 13.1** sorts the suffixes with `string::compare`, computes each LCP by direct comparison, and prints the table of **Figure 13.1** together with the count of new prefixes per row.

Listing 13.1 `naive_sa.cpp`: the suffix array and LCP array computed the slow way, printed as a table.

```
// Warm-up: build the suffix array and the LCP column the slow, obvious way
// (sort whole suffixes as strings) and print them as a table.
#include <algorithm>
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

int main() {
    char buf[1001];
    if (scanf("%1000s", buf) != 1) return 0;
    string s = buf;
    int n = (int)s.size();
    vector<int> sa(n);
    for (int i = 0; i < n; i++) sa[i] = i;
    sort(sa.begin(), sa.end(), [&](int a, int b) {
        return s.compare(a, string::npos, s, b, string::npos) < 0;
    });
    printf(" k  sa  lcp  new  suffix\n");
    long long distinct = 0;
    for (int k = 0; k < n; k++) {
        int h = 0;
        if (k > 0)
            while (sa[k] + h < n && sa[k - 1] + h < n &&
                   s[sa[k] + h] == s[sa[k - 1] + h]) h++;
        int fresh = n - sa[k] - h; // prefixes not seen before
        distinct += fresh;
        printf("%2d %2d %3d %3d %s\n", k, sa[k], h, fresh, s.c_str() + sa[k]);
    }
    printf("distinct substrings: %lld\n", distinct);
    return 0;
}
```

For the input ABABA:

```
k  sa  lcp  new  suffix
0  4   0   1   A
1  2   1   2  ABA
2  0   3   2  ABABA
3  3   0   2   BA
4  1   2   2  BABA
distinct substrings: 9
```

This program is $O(n^2 \log n)$, which is fine for a few thousand characters and a perfect reference for testing.

13.3.2 Pattern search with the suffix array

Because the suffixes are sorted, all suffixes that start with a pattern p sit next to each other in the suffix array; they form one block, just as all words starting with “ban” sit together in a dictionary.

Two binary searches find the ends of the block, and its size is the number of occurrences. **Listing 13.2** does exactly that, comparing only the first $|p|$ characters of each suffix.

Listing 13.2 `pattern_count.cpp`: counting occurrences of a pattern by binary search on the suffix array.

```
// Warm-up: count the occurrences of a pattern p in a text s by binary search
// on the suffix array (here built naively): the suffixes that start with p
// form one contiguous block of the sorted order.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
    string s, p;
    if (!(cin >> s >> p)) return 0;
    int n = (int)s.size();
    vector<int> sa(n);
    for (int i = 0; i < n; i++) sa[i] = i;
    sort(sa.begin(), sa.end(), [&](int a, int b) {
        return s.compare(a, string::npos, s, b, string::npos) < 0;
    });
    // compare only the first |p| characters of each suffix with p
    auto below = [&](int start, const string& q) {
        return s.compare(start, q.size(), q) < 0;
    };
    auto above = [&](const string& q, int start) {
        return s.compare(start, q.size(), q) > 0;
    };
    auto lo = lower_bound(sa.begin(), sa.end(), p, below);
    auto hi = upper_bound(sa.begin(), sa.end(), p, above);
    cout << p << " occurs " << (hi - lo) << " time(s) at:";
    vector<int> at(lo, hi);
    sort(at.begin(), at.end());
    for (int i : at) cout << ' ' << i;
    cout << '\n';
    return 0;
}
```

For the text ABRACADABRA and the pattern ABRA:

```
ABRA occurs 2 time(s) at: 0 7
```

Each search costs $O(|p|\log n)$ once the suffix array is built, no matter how many times the pattern occurs. Once a text is indexed, you can answer thousands of pattern queries quickly.

13.4 Case Study: DISUBSTR

13.4.1 The problem in our own words

The SPOJ problem DISUBSTR [4] gives $T \leq 20$ test cases. Each is a string of at most 1000 characters without spaces. For each string, print the number of its distinct nonempty substrings.

13.4.2 Sample

```
2
CCCCC
ABABA
```

5
9

These are the two strings we already analyzed: CCCCC has five distinct substrings (C, CC, CCC, CCCC, CCCCC) and ABABA has nine.

Before designing, it helps to see how small the numbers are. With $T = 20$ strings of 1000 characters, even an $O(n^2)$ method performs about $2 \cdot 10^7$ steps, so the suffix array is not *required* for speed here. It is the right choice anyway, because the same code solves the harder versions of the problem, and because DISUBSTR is the natural place to learn it on inputs that are small enough to check by hand and by brute force.

Our own extra tests are just as instructive. The string A has one distinct substring. The string AB has three: A, B and AB. The string AAB has five: A, B, AA, AB and AAB; its sorted suffixes are AAB, AB, B with LCP values 1 and 0, and [Equation \(13.4\)](#) gives $6 - 1 = 5$. Working a few such strings by hand before coding is the quickest way to make sure the formula, and not only the program, is understood.

13.4.3 Design

The design is [Equation \(13.4\)](#): build the suffix array by doubling (**Pseudocode 13.1**), compute the LCP array by Kasai, and subtract the LCP sum from $n(n + 1)/2$. Two small decisions matter.

- **Character codes.** The initial rank is `(unsigned char)s[i]`, so every byte value is a valid, non-negative rank and the out-of-range marker -1 is smaller than all of them.
- **64-bit total.** For $n = 1000$ the total $n(n + 1)/2 = 500\,500$ fits in `int`, but the same code applied to $n = 10^5$ (Exercise 5) would reach $5 \cdot 10^9$. We use `long long` from the start.

13.4.4 The complete solution

Listing 13.3 is the full program. Building the suffix array is the first half of the core; the Kasai loop, between the markers, is the second.

Listing 13.3 `DISUBSTR.cpp`: distinct substrings with a suffix array (prefix doubling) and Kasai's LCP.

```
// DISUBSTR: count the distinct substrings of a string.
// Suffix array by prefix doubling, LCP array by Kasai, then
// answer = n(n+1)/2 - (sum of the LCPs of neighboring suffixes).
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// Suffix array: sa[k] is the start of the k-th smallest suffix of s.
vector<int> suffixArray(const string& s) {
    int n = (int)s.size();
    vector<int> sa(n), rk(n), tmp(n);
    for (int i = 0; i < n; i++) sa[i] = i, rk[i] = (unsigned char)s[i];
    for (int gap = 1; gap < n; gap *= 2) {
        // sort by the pair (rank of first gap chars, rank of the next gap)
        auto key2 = [&](int i) { return i + gap < n ? rk[i + gap] : -1; };
        auto less = [&](int a, int b) {
            if (rk[a] != rk[b]) return rk[a] < rk[b];
            return key2(a) < key2(b);
        };
        sort(sa.begin(), sa.end(), less);
        tmp[sa[0]] = 0; // equal pairs share a rank
        for (int k = 1; k < n; k++)
            tmp[sa[k]] = tmp[sa[k - 1]] + (less(sa[k - 1], sa[k]) ? 1 : 0);
        rk = tmp;
        if (rk[sa[n - 1]] == n - 1) break; // all ranks distinct: done
    }
    return sa;
}

// lcp[k] = length of the longest common prefix of suffixes sa[k-1] and sa[k].
vector<int> lcpArray(const string& s, const vector<int>& sa) {
    int n = (int)s.size();
    vector<int> pos(n), lcp(n, 0);
    for (int k = 0; k < n; k++) pos[sa[k]] = k;
    // <P50>
    // Kasai: visit suffixes in text order i = 0, 1, ...; compare suffix i with
    // its left neighbor in sa, reusing h, which drops by at most 1 per step.
    // >>> YOUR TURN (Public Edition): write this part yourself, following the hint above.
    <<<
    // </P50>
    return lcp;
}
```

Listing 13.3 (Continued) DISUBSTR.cpp: distinct substrings with a suffix array (prefix doubling) and Kasai's LCP.

```
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;
    while (t--) {
        string s;
        cin >> s;
        long long n = (long long)s.size();
        vector<int> sa = suffixArray(s);
        vector<int> lcp = lcpArray(s, sa);
        long long total = n * (n + 1) / 2;
        for (int v : lcp) total -= v;
        cout << total << '\n';
    }
    return 0;
}
```

13.4.5 Line by line

suffixArray. The vectors `sa`, `rk` and `tmp` hold the current order, the current ranks and the next ranks. The lambda `key2` returns the second component of the pair in Equation (13.3), with -1 past the end. The comparator `less` compares pairs lexicographically. After `sort`, the loop over `k` assigns new ranks: the first suffix gets 0, and each next suffix gets the previous rank plus one exactly when its pair is strictly larger, which is what `less(sa[k - 1], sa[k])` tests. The early exit checks whether the last suffix has rank $n - 1$; since ranks increase by at most one per step, that happens only when all n ranks are different, and further rounds cannot change the order.

lcpArray. `pos` is the inverse of `sa`: `pos[i]` is the place of suffix i in sorted order. The loop visits $i = 0, 1, \dots$ in text order. The smallest suffix has no predecessor, so `h` is reset and the loop moves on. Otherwise `j` is the predecessor, the `while` loop extends the common prefix starting from the current `h`, the result is stored at `lcp[pos[i]]`, and `h` drops by one for the next step, as allowed by Kasai's lemma.

main. For each string it computes the arrays and prints $n(n + 1)/2 - \sum lcp$. Note that `lcp[0]` is 0, so summing the whole vector is correct.

TRAP. (1) Forgetting the -1 for positions past the end and reading `rk[i + gap]` out of bounds. (2) Assigning new ranks from the sorted position `k` instead of comparing neighbor pairs: equal pairs must share a rank, or the next round sorts by an order that does not exist. (3) Resetting `h` to 0 at every `i` in Kasai: still correct, but $O(n^2)$. (4) Skipping the smallest suffix without setting `h = 0`: Kasai's lemma gives no lower bound after a suffix that has no predecessor, and a leftover `h` would skip comparisons and store an LCP that is too large. (5) Reading strings with `scanf("%s")` into a buffer that is one character too short for the terminating zero.

13.5 Testing It: Suffix Array Against a Set of Strings

The reference for DISUBSTR is almost a definition: put every substring into a `set<string>` and print its size (Listing 13.4). It is $O(n^3 \log n)$ and happily handles strings of 40 characters.

Listing 13.4 `brute.cpp`: the set-of-substrings reference.

```
// DISUBSTR reference: put every substring into a set and count the set.
#include <iostream>
#include <set>
#include <string>
using namespace std;

int main() {
    int t;
    if (!(cin >> t)) return 0;
    while (t--) {
        string s;
        cin >> s;
        set<string> seen;
        for (size_t i = 0; i < s.size(); i++)
            for (size_t len = 1; i + len <= s.size(); len++)
                seen.insert(s.substr(i, len));
        cout << seen.size() << '\n';
    }
    return 0;
}
```

The generator (**Listing 13.5**) mixes two alphabets. Strings over only A and B are highly repetitive, which exercises long LCPs and many doubling rounds; strings over a mixed alphabet that includes lowercase letters and digits check that the character codes are handled correctly.

Listing 13.5 `gen.py`: random strings over a small and a mixed alphabet.

```
# Random DISUBSTR input: short strings over a small alphabet (many repeats).
import random
import sys

rnd = random.Random(int(sys.argv[1]))
t = rnd.randint(1, 5)
print(t)
for _ in range(t):
    alpha = "AB" if rnd.random() < 0.5 else "ABCZaz09"
    n = rnd.randint(1, 40)
    print("".join(rnd.choice(alpha) for _ in range(n)))
```

With the usual `stress.sh` driver:

```
All 500 tests passed.
```

The sanitizers (`-fsanitize=address,undefined`) report nothing on these inputs either, which confirms that the `-1` guard in `key2` keeps every index in range.

13.6 Contest-Day Tactics

Table 13.1 shows that DISUBSTR at its limits is no challenge for time; the challenge is correctness.

Table 13.1 Running time of `DISUBSTR.cpp` at the maximum size.

Input	Time
20 random strings of 1000 letters from A–J	0.01 s
20 strings of 1000 copies of A	0.01 s

- **Keep a tested suffix array in your memory, not in your notes only.** In OSN you type everything yourself. Practice writing **Pseudocode 13.1** and Kasai until it takes under ten minutes, then check it against `naive_sa.cpp`.
- **Know the dictionary of uses.** With (sa, lcp) in hand: the longest repeated substring is $\max lcp$; the number of distinct substrings is **Equation (13.4)**; pattern search is a binary search; the k -th smallest distinct substring is found by walking the sorted suffixes and spending the new prefixes $n - sa[k] - lcp[k]$ of each; the longest common substring of two strings comes from the suffix array of $s \# t$ [5], [6].
- **Choose the simplest tool that fits.** For $n \leq 1000$ even `naive_sa.cpp` would pass DISUBSTR: $O(n^2 \log n)$ with fast comparisons. A hash of all substrings into a set would also pass. The suffix array pays off when n reaches 10^5 , as in the sister problem SUBST1 (Exercise 5).

13.6.1 Two strings at once

Many string problems involve two texts: find the longest piece that both essays contain, or count the pieces that appear in one but not the other. The standard trick is to glue them into one string $u = s \# t$, where $\#$ is a separator that occurs in neither text and sorts before every letter. Build the suffix array and LCP array of u . Every suffix now belongs to s (it starts before the $\#$) or to t (it starts after it). The longest common substring of s and t is the largest $lcp[k]$ among neighbors $sa[k - 1]$ and $sa[k]$ that belong to *different* texts. The separator guarantees that no common

prefix runs across the border between the two texts, so every LCP between an s -suffix and a t -suffix is a real common substring.

For example, with $s = \text{ABAB}$ and $t = \text{BABA}$ the glued string is $\text{ABAB\#BABA}$. Among its sorted suffixes, ABA (from t) and $\text{ABAB\#BABA}$ (from s) are neighbors with LCP 3, and $\text{BAB\#BABA}$ (from s) and BABA (from t) are neighbors with LCP 3 as well, so the longest common substring has length 3: ABA or BAB . The whole computation is one suffix array of length $|s| + |t| + 1$.

13.6.2 Choosing the right tool

A contestant who knows several string tools should know when each one pays off. *Polynomial hashing* compares two substrings in $O(1)$ after $O(n)$ preparation and is quick to write; it is ideal for “is this substring equal to that one?” and for binary searching a length, but its answers are correct only with high probability, and counting distinct substrings with hashes needs a set of $O(n^2)$ hash values, fine for $n = 1000$, hopeless for $n = 10^5$. The *suffix array with LCP* is deterministic, uses little memory, and answers everything that depends on the sorted order of suffixes: distinct substrings, the k -th substring, longest repeats, common substrings of two texts. The *suffix automaton* and the *suffix tree* can do even more online, but they are longer and harder to write under pressure [6]. For an olympiad, a well-practiced suffix array plus hashing covers almost every string task you will meet.

13.6.3 Common mistakes

Three mistakes are typical of first suffix array implementations. Mixing up sa and its inverse: $sa[k]$ answers “which suffix is k -th?”, while $pos[i]$ answers “where is suffix i ?”, and Kasai needs both. Comparing *characters* instead of *ranks* in later rounds of doubling, which silently sorts only by the first character. And forgetting that the input may contain characters other than capital letters: using `s[i] - 'A'` as a rank breaks on lowercase letters or digits, while the unsigned character code always works.

13.7 Chapter Summary

- Every substring is a prefix of a suffix; sorting the suffixes (Equation (13.1)) organizes all substrings.
- Prefix doubling sorts suffixes by rank pairs (Equation (13.3)), doubling the compared length each round: $O(n \log^2 n)$ with a comparison sort.
- Kasai’s lemma lets the LCP of neighbors (Equation (13.2)) be computed in $O(n)$ by visiting suffixes in text order.
- Distinct substrings $= n(n + 1)/2 - \sum lcp$, Equation (13.4), because each suffix contributes only the prefixes longer than its LCP with its predecessor.
- A set of all substrings is a perfect reference for stress testing on short strings.

13.8 Exercises

1. Build sa and lcp for BANANA by hand and find the number of distinct substrings.
2. Using the arrays from Exercise 1, find the longest substring of BANANA that occurs at least twice.

3. How many distinct substrings does a string of n equal characters have? And a string of n different characters? Check both with [Equation \(13.4\)](#).
4. Describe how to find the k -th smallest distinct substring in lexicographic order using *sa* and *lcp*.
5. SPOJ SUBST1 [7] is DISUBSTR with strings of up to 50 000 characters. Which parts of `DISUBSTR.cpp` must change, if any?
6. Why does **Pseudocode 13.1** treat a position past the end of the string as rank -1 rather than as rank 0?
7. Using the suffix array of s , how would you find the length of the longest substring that occurs at least k times?
8. A string s has 1000 characters and all its LCP values are 0. What can you say about s ?

13.9 Solutions to the Exercises

1. The suffixes of BANANA (positions 0–5) in sorted order are A (5), ANA (3), ANANA (1), BANANA (0), NA (4), NANA (2). So $sa = [5, 3, 1, 0, 4, 2]$ and $lcp = [0, 1, 3, 0, 0, 2]$. The count is $21 - 6 = 15$.
2. The longest repeated substring is the one with the largest LCP: $lcp[2] = 3$, between ANA and ANANA, so the answer is ANA, which occurs at positions 1 and 3 (the occurrences overlap).
3. For n equal characters the sorted suffixes are the lengths $1, 2, \dots, n$ and $lcp[k] = k$, so the count is $n(n+1)/2 - n(n-1)/2 = n$. For n different characters all LCPs are 0 and the count is $n(n+1)/2$.
4. Walk $k' = 0, 1, \dots$ through the sorted suffixes. Suffix $sa[k']$ contributes the new substrings of lengths $lcp[k'] + 1$ to $n - sa[k']$, in increasing order, and all of them come after every substring contributed earlier. Subtract $n - sa[k'] - lcp[k']$ from k until k fits inside the current suffix's contribution; the answer is then the prefix of length $lcp[k'] + k$ of suffix $sa[k']$.
5. The algorithm stays. With $n = 50\,000$ the doubling costs about $50\,000 \cdot 16 \cdot 16 \approx 1.3 \cdot 10^7$ comparisons per string, fine for a handful of strings. The answer can reach $1.25 \cdot 10^9$, which still fits in `long long` (and the code already uses it). If many long strings appear, replace `sort` by radix sort to get $O(n \log n)$.
6. A suffix that ends before the compared length is a proper prefix of the other suffix's first $2g$ characters and must sort *before* it. Rank 0 is already used by real characters and would make "A" and "A followed by the smallest character" look equal; -1 is smaller than every real rank.
7. A substring occurs at least k times exactly when it is a common prefix of k suffixes that are consecutive in the suffix array. The longest common prefix of k consecutive suffixes $sa[i], \dots, sa[i+k-1]$ is the minimum of $lcp[i+1], \dots, lcp[i+k-1]$. So the answer is the maximum, over all windows of $k-1$ consecutive LCP values, of the window minimum. A sliding-window minimum with a deque computes it in $O(n)$ after the arrays are built; for $k = 2$ it reduces to $\max lcp$, the longest repeated substring of Section 13.6.
8. Neighboring suffixes in sorted order share no first character, so no character repeats: if some character appeared twice, the two suffixes starting with it would be consecutive or have only suffixes starting with the same character between them, giving a positive LCP. So all 1000

characters are different, which is possible only with an alphabet of at least 1000 symbols, and the answer to DISUBSTR is $n(n + 1)/2 = 500\,500$.

References

- [1] 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.
- [2] J. Kärkkäinen, P. Sanders, and S. Burkhardt, “Linear work suffix array construction,” *Journal of the ACM*, vol. 53, no. 6, pp. 918–936, 2006.
- [3] 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)*, LNCS 2089, Springer, 2001, pp. 181–192.
- [4] Sphere Online Judge, “DISUBSTR - Distinct Substrings.” [Online]. Available: <https://www.spoj.com/problems/DISUBSTR/>
- [5] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 2. Lulu Press, 2020.
- [6] cp-algorithms.com, “Suffix Array.” [Online]. Available: <https://cp-algorithms.com/string/suffix-array.html>
- [7] Sphere Online Judge, “SUBST1 - New Distinct Substrings.” [Online]. Available: <https://www.spoj.com/problems/SUBST1/>

Chapter 14

Finite Differences, Interpolation and Modular Arithmetic — BATTLECRY

Some counting problems look frightening: count the positions of four chess pieces on an $N \times N$ board that satisfy a subtle rule, for N up to a hundred thousand. No search can visit those positions, and a hand-derived formula would take hours and invite mistakes. Yet many such problems share a hidden simplicity: once N is large enough, the answer is a *polynomial* in N . This chapter teaches a mechanical pipeline for that situation. Write an honest brute force for small N ; detect the degree with a *finite-difference table*; recover the polynomial by *Lagrange interpolation*; and evaluate it for huge N with *Horner's rule* and *modular inverses*. You will apply the pipeline to the SPOJ problem BATTLECRY, where two rooks and a king checkmate a lone king, and end with an $O(1)$ solution whose correctness you have checked against the brute force yourself.

14.1 Where This Helps: Counting Without Listing

Pak Hadi builds square tile floors for mosques and houses. A customer asks him how many ways there are to place four decorative tiles on an $N \times N$ floor so that a certain pattern rule holds. For $N = 4$ he can draw them all on graph paper; for $N = 40$ he cannot. But he notices that when he counts for $N = 4, 5, 6, 7$, the numbers grow smoothly, like the area or the volume of something. If the count really is a polynomial, a handful of small cases determine it completely, and then any N is one formula away.

The same phenomenon appears whenever objects are placed on a grid with a *fixed number* of pieces and *local* rules: counting non-attacking queens or kings of a fixed number, counting rectangles or triangles of a certain kind in an $N \times N$ grid, counting paths of fixed length in a growing structure. In olympiad problems it usually shows up as “for many values of N , print the answer modulo a prime”. The pipeline of this chapter turns such a problem into an afternoon of careful brute force followed by a few lines of arithmetic.

It helps to be honest about what the pipeline is. It is not a proof that the formula is right; it is an experiment followed by a careful check. In a contest that is exactly what we need: the judge only tests whether the answers are correct, and the combination of a trustworthy brute force, a clearly constant difference row and extra validation points makes a wrong formula extremely unlikely. After the contest, if you are curious, you can try to derive the formula by hand from the shapes of Section 14.2.4, and it is a rewarding exercise in careful counting.

KEY INSIGHT. A small brute force is not wasted work. It is *data*. When the answer is a polynomial in N , the brute force for small N contains the whole solution, and finite differences and interpolation extract it.

14.2 The Technique from Zero

14.2.1 Finite differences: the polynomial stethoscope

Given a sequence of values $f(3), f(4), f(5), \dots$, its *first difference* is the sequence

$$\Delta f(N) = f(N + 1) - f(N) \quad (14.1)$$

and the d -th difference is $\Delta^d f = \Delta(\Delta^{d-1} f)$. The key property is:

THEOREM. If f is a polynomial of degree d with leading coefficient a , then Δf is a polynomial of degree $d - 1$, the d -th difference is the constant $a \cdot d!$, and all higher differences are 0. Conversely, if the d -th differences of the values $f(N_0), f(N_0 + 1), \dots$ are constant, then those values agree with a polynomial of degree d [1].

Why. For $f(N) = aN^d + \dots$, the difference $f(N + 1) - f(N)$ contains $a((N + 1)^d - N^d) = adN^{d-1} + \dots$: the degree drops by one and the leading coefficient is multiplied by the old degree. Applying this d times leaves the constant $a \cdot d \cdot (d - 1) \cdots 1 = a \cdot d!$.

So the procedure for a mystery sequence is: write the values in a row, write their differences in the row below, and repeat. If some row becomes constant, its index is the degree. **Figure 14.1** shows the table for the answer of this chapter's problem, computed by the brute force in Section 14.5: the fifth row is the constant 480.

	N=3	N=4	N=5	N=6	N=7	N=8	N=9	N=10
row 0	232	1432	5188	14536	34464	72392	138652	246968
row 1		1200	3756	9348	19928	37928	66260	108316
row 2			2556	5592	10580	18000	28332	42056
row 3				3036	4988	7420	10332	13724
row 4					1952	2432	2912	3392
row 5						480	480	480

← constant row: degree 5

Figure 14.1 The finite-difference table of the BATTLECRY counts for $N = 3$ to 10. Each row holds the differences of the row above; row 5 is constant, so the counts follow a polynomial of degree 5.

From the theorem, $480 = a \cdot 5!$, so the leading coefficient is $a = 480/120 = 4$: for large boards the count grows like $4N^5$.

14.2.2 Lagrange interpolation: from points to a polynomial

A polynomial of degree d is determined by its values at $d + 1$ different points. Lagrange's formula writes it down directly. Given points $(x_0, y_0), \dots, (x_d, y_d)$ with distinct x_i ,

$$P(x) = \sum_{i=0}^d y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \quad (14.2)$$

Each product in Equation (14.2) is a polynomial of degree d that equals 1 at x_i and 0 at every other x_j , so $P(x_i) = y_i$ for every i . Uniqueness follows because the difference of two such polynomials would have degree at most d and $d + 1$ roots, which forces it to be zero [2].

We can use Equation (14.2) in two ways. We can evaluate it directly at a new x , which needs no coefficients at all; or we can expand it once, on paper or with a small script using exact fractions, to get the coefficients. For BATTLECRY, interpolating through the six points $N = 3, \dots, 8$ and expanding gives

$$f(N) = 4N^5 - \frac{56}{3}N^4 + 22N^3 + \frac{554}{3}N^2 - 764N + 808, \quad N \geq 3 \quad (14.3)$$

The leading coefficient 4 agrees with the difference table. Two coefficients are fractions, so we multiply by 3 to work with integers:

$$3f(N) = 12N^5 - 56N^4 + 66N^3 + 554N^2 - 2292N + 2424 \quad (14.4)$$

and check Equation (14.4) against the brute-force values for $N = 9$ and $N = 10$, which were *not* used in the interpolation: both agree (138 652 and 246 968). That check matters. The difference table is strong evidence but not a proof, and the extra points guard against a pattern that changes later.

14.2.3 Horner's rule and modular arithmetic

The answer must be printed modulo the prime $M = 7\,340\,033$. Two tools are needed.

Horner's rule evaluates a polynomial with one multiplication and one addition per coefficient, by nesting:

$$\begin{aligned} 12N^5 - 56N^4 + 66N^3 + 554N^2 - 2292N + 2424 \\ = (((((12N - 56)N + 66)N + 554)N - 2292)N + 2424 \end{aligned} \quad (14.5)$$

Taking the remainder after every step keeps all intermediate values below $M^2 \approx 5.4 \cdot 10^{13}$, far inside the 64-bit range.

Division modulo a prime is multiplication by an *inverse*. We cannot divide the remainder of $3f(N)$ by 3 with $/$, because the remainder need not be a multiple of 3. Instead we multiply by the number 3^{-1} that satisfies $3 \cdot 3^{-1} \equiv 1 \pmod{M}$. Fermat's little theorem says $a^{M-1} \equiv 1 \pmod{M}$ for a prime M and a not divisible by M , so

$$a^{-1} \equiv a^{M-2} \pmod{M} \quad (14.6)$$

and fast exponentiation computes Equation (14.6) in $O(\log M)$ multiplications [3]. For $a = 3$ the result is $3^{-1} = 2\,446\,678$; indeed $3 \cdot 2\,446\,678 = 7\,340\,034 = M + 1$.

14.2.4 Why placements give polynomials

Why should a chess count be a polynomial at all? Classify every position by its *shape*: which pieces share a row or a column, the exact small offsets between pieces that are close together, and which pieces touch the border of the board. For a fixed shape, the number of positions is a product of factors like $(N - c)$, one for each free coordinate, so it is a polynomial in N . The rule “the black king is checkmated” depends only on the shape, because it involves only lines, adjacency and borders. Once N is large enough for every shape to fit, the total is a sum of finitely many polynomials, hence a polynomial. For small N some shapes do not fit, which is why the formula starts only at some N_0 ; here $N_0 = 3$, and $f(1) = f(2) = 0$, while Equation (14.3) would give other values. The shape argument also bounds the degree: there are 8 coordinates, and each constraint that ties a coordinate to another piece or to the border removes one degree of freedom. The table then tells us the exact degree, 5.

14.2.5 Fast exponentiation by hand

Fermat's formula Equation (14.6) needs a power with a huge exponent, and fast exponentiation computes it with repeated squaring. Take the tiny case $3^{-1} \bmod 7 = 3^5 \bmod 7$. In binary, $5 = 101_2$. Start with result $r = 1$ and base $b = 3$.

- Lowest bit of 5 is 1: $r = 1 \cdot 3 = 3$. Square the base: $b = 9 \bmod 7 = 2$. Shift: the exponent becomes 2.
- Lowest bit of 2 is 0: r stays 3. Square: $b = 4$. The exponent becomes 1.
- Lowest bit of 1 is 1: $r = 3 \cdot 4 = 12 \bmod 7 = 5$. The exponent becomes 0 and we stop.

So $3^{-1} \equiv 5 \pmod{7}$, and indeed $3 \cdot 5 = 15 \equiv 1$. For $M = 7\,340\,033$ the exponent $M - 2$ has 23 bits, so `power` performs at most 23 squarings and 23 multiplications. Every product is of two numbers below M , so it stays below $5.4 \cdot 10^{13}$, safely inside `long long`.

14.2.6 How many values do we need?

For a polynomial of degree d we need $d + 1$ values to determine it, and at least two or three more to see that the d -th differences are constant and to validate the result. Since we usually do not know d in advance, the brute force should produce as many values as it can within a few minutes of running time. For BATTLECRY the brute force costs about $N^8/2$ checks, so $N = 10$ (about $5 \cdot 10^7$ positions) is comfortable and $N = 14$ (about $7 \cdot 10^8$) is still possible if we are patient. With values for $N = 3, \dots, 10$ we have eight points: six to interpolate a quintic and two to check it.

It also pays to think about the degree *before* computing. Each free coordinate of a piece contributes at most one factor N . Here one rook must share the black king's row or column to give check, and the other rook and the white king must stand in tight relations to the king to cover its escape squares. Such constraints remove degrees of freedom, so the degree should be well below the 8 coordinates, and 5 is plausible. A prediction that disagrees with the difference table is a warning that either the brute force or the reasoning is wrong.

14.3 Warm-Up Problems

14.3.1 The difference table

Listing 14.1 prints the table of **Figure 14.1** and stops at the first constant row. The values are the brute-force counts for $N = 3$ to 10.

Listing 14.1 `diff_table.cpp`: the finite-difference table and the detected degree.

```
// Warm-up: print the finite-difference table of f(3), f(4), ..., f(10) and
// report the first row that is constant: that row's index is the degree.
#include <cstdio>
#include <vector>
using namespace std;

int main() {
    vector<long long> row = {232, 1432, 5188, 14536,
                           34464, 72392, 138652, 246968};
    for (int d = 0; !row.empty(); d++) {
        printf("row %d:", d);
        bool constant = true;
        for (size_t i = 0; i < row.size(); i++) {
            printf(" %lld", row[i]);
            if (row[i] != row[0]) constant = false;
        }
        printf("\n");
        if (constant && row.size() > 1) {
            printf("row %d is constant -> degree %d\n", d, d);
            break;
        }
        vector<long long> next;
        for (size_t i = 0; i + 1 < row.size(); i++)
            next.push_back(row[i + 1] - row[i]);
        row = next;
    }
    return 0;
}
```

```
row 0: 232 1432 5188 14536 34464 72392 138652 246968
row 1: 1200 3756 9348 19928 37928 66260 108316
row 2: 2556 5592 10580 18000 28332 42056
row 3: 3036 4988 7420 10332 13724
row 4: 1952 2432 2912 3392
row 5: 480 480 480
row 5 is constant -> degree 5
```

With eight values we can see a constant row of length 3 at depth 5. In practice, compute a few more values than the degree you expect, so that the constant row has at least two or three entries; a “constant” row of length 1 proves nothing.

14.3.2 Lagrange evaluation modulo a prime

Listing 14.2 evaluates Equation (14.2) directly, modulo $10^9 + 7$, for points at $x = 1, \dots, k$. The example sequence 1, 5, 14, 30 is the sums of squares $1^2 + \dots + n^2$, a cubic polynomial, so four points suffice.

Listing 14.2 `lagrange.cpp`: evaluating the interpolating polynomial at a new point without computing coefficients.

```
// Warm-up: given f(1), ..., f(k) of a polynomial of degree < k, evaluate f(x)
// modulo a prime with the Lagrange formula (no coefficients needed).
#include <cstdio>
#include <vector>
using namespace std;

const long long MOD = 1000000007;

long long power(long long b, long long e) {
    long long r = 1;
    b %= MOD;
    for (; e > 0; e >>= 1, b = b * b % MOD)
        if (e & 1) r = r * b % MOD;
    return r;
}

long long interpolate(const vector<long long>& y, long long x) {
    int k = (int)y.size(); // points are (1, y[0]) .. (k, y[k-1])
    long long total = 0;
    for (int i = 1; i <= k; i++) {
        long long num = 1, den = 1; // basis polynomial L_i(x)
        for (int j = 1; j <= k; j++) {
            if (j == i) continue;
            num = num * (((x - j) % MOD + MOD) % MOD) % MOD;
            den = den * (((i - j) % MOD + MOD) % MOD) % MOD;
        }
        long long term = y[i - 1] % MOD * num % MOD * power(den, MOD - 2) % MOD;
        total = (total + term) % MOD;
    }
    return total;
}

int main() {
    vector<long long> y = {1, 5, 14, 30}; // f(1..4): sums of squares
    for (long long x : {5LL, 10LL, 100LL})
        printf("f(%lld) = %lld\n", x, interpolate(y, x));
    return 0;
}
```

```
f(5) = 55
f(10) = 385
f(100) = 338350
```

The known closed form $n(n+1)(2n+1)/6$ gives 55, 385 and 338 350 as well. Each evaluation costs $O(k^2 + k \log M)$; with the classical prefix and suffix products it can be made $O(k)$ when the points are consecutive integers.

14.3.3 The whole pipeline on a smaller problem

Before the chess problem, run the pipeline on one whose answer we can also derive by hand: *in how many ways can two identical kings be placed on an $N \times N$ board so that they do not attack each other?* A brute force (or careful counting on paper) gives, for $N = 1, \dots, 6$:

0, 0, 16, 78, 228, 520

The differences are 0,16,62,150,292; then 16,46,88,142; then 30,42,54; then 12,12. The fourth row is constant, so the degree is 4 and the leading coefficient is $12/4! = 1/2$. Five points determine a quartic, and interpolation gives

$$g(N) = \frac{N^2(N^2 - 1)}{2} - 2N(N - 1) - 2(N - 1)^2$$

which is exactly what direct counting says: all $\binom{N^2}{2}$ pairs of squares, minus the $N(N-1)$ horizontal, $N(N-1)$ vertical and $2(N-1)^2$ diagonal neighbor pairs. Unlike BATTLECRY, this polynomial is valid from $N = 1$: the shape argument of Section 14.2.4 predicts that, because every shape (two squares side by side, one above the other, or diagonal) fits as soon as the board is 2×2 , and for $N = 1$ every term is 0 anyway. The pipeline did not need the insight, but the insight confirms the pipeline, and in a contest that double check is worth a minute.

14.4 Case Study: BATTLECRY

14.4.1 The problem in our own words

The SPOJ problem BATTLECRY [4] places four pieces on an $N \times N$ chessboard: a white king, a black king and two identical white rooks, all on different squares. Count the positions in which the two kings are not on adjacent squares and the black king, whose side is to move, is checkmated: it is attacked by a rook and has no legal move. There are many test cases, each giving one N (up to about 10^5); for test case k print **Case k:** followed by the count modulo 7 340 033.

14.4.2 Understanding checkmate precisely

A brute force must implement the rules exactly, so it is worth stating them.

- A rook attacks along its row and column until the first occupied square. The white king and the other rook block its line; the black king does not block squares *behind itself* when it moves, because it is the piece that moves away.
- The black king is *in check* if a rook attacks its square.
- A king move to a neighboring square t is legal if t is on the board, t is not adjacent to the white king, and no remaining rook attacks t . If t holds a rook, the move captures it, and only the other rook can attack t .
- *Checkmate* means in check and no legal move.

A mate on the smallest board. Number rows and columns from 0 on a 3×3 board. Put the black king in the corner (0,0), a rook on (0,2) and the other rook on (1,2), and the white king on (2,0). The kings are two rows apart, so they are not adjacent. The rook on (0,2) attacks along row 0 through the empty square (0,1) to the king: check. The king's three neighbors are all covered: (0,1) by the first rook, (1,0) and (1,1) by the second rook along row 1, and both of these also by the white king. Checkmate, and one of the 232 positions counted for $N = 3$. Move the first rook to (0,1) instead and the king simply captures it, because no white piece protects that square: this is why the brute force must treat captures carefully. Such small hand-checked positions are the best test that the rule code in the brute force means what we think it means.

14.4.3 The pipeline, step by step

1. **Brute force.** Enumerate every position for $N \leq 10$ and count checkmates (Section 14.5). The counts for $N = 1, 2, \dots, 10$ are 0, 0, 232, 1432, 5188, 14536, 34464, 72392, 138652 and 246968.
2. **Detect the degree.** The difference table of the values from $N = 3$ (**Figure 14.1**) has constant fifth differences: degree 5.
3. **Interpolate.** Six points, $N = 3..8$, give **Equation (14.3)**; multiplying by 3 gives the integer coefficients of **Equation (14.4)**.
4. **Validate.** $N = 9$ and $N = 10$ agree with the formula.

5. Evaluate fast. For each query: 0 if $N < 3$; otherwise Horner (Equation (14.5)) modulo M , then multiply by 3^{-1} .

14.4.4 Sample

Our own test input with five queries, and the output of the solution:

```
5
1
3
4
8
100000
```

```
Case 1: 0
Case 2: 232
Case 3: 1432
Case 4: 72392
Case 5: 4690063
```

The first four values are the brute-force counts; the last is $f(100\,000) \bmod 7\,340\,033$, which we also checked with exact big-integer arithmetic in Python.

14.4.5 The complete solution

Listing 14.3 is the whole program. All the thinking happened on paper and in the brute force; the code is short. The second half of the evaluation, the last four Horner steps and the division by 3, lies between the markers.

Listing 14.3 `BATTLECRY.cpp`: the degree-5 formula evaluated with Horner's rule and a modular inverse.

```
// BATTLECRY: checkmates with a white king and two white rooks against a Lone
// black king on an N x N board, modulo 7340033. For N >= 3 the count is
// f(N) = (12N^5 - 56N^4 + 66N^3 + 554N^2 - 2292N + 2424) / 3; below 3 it is 0.
#include <cstdio>
using namespace std;

const long long MOD = 7340033; // a prime: 7 * 2^20 + 1

long long power(long long b, long long e) { // b^e mod MOD, fast exponentiation
    long long r = 1;
    b %= MOD;
    while (e > 0) {
        if (e & 1) r = r * b % MOD;
        b = b * b % MOD;
        e >>= 1;
    }
    return r;
}

const long long INV3 = power(3, MOD - 2); // Fermat: 3 * INV3 = 1 (mod MOD)

long long mates(long long n) {
    if (n < 3) return 0; // the polynomial starts at N = 3
    long long v = n % MOD;
    long long h = 12; // Horner, highest coefficient first
    h = (h * v % MOD - 56 + MOD) % MOD;
    // <P50>
    // Continue Horner with the coefficients 66, 554, -2292, 2424 (keep every
    // step in [0, MOD) by adding MOD before %), then divide by 3 via INV3.
    // >>> YOUR TURN (Public Edition): write this part yourself, following the hint above.
    <<<
    // </P50>
}

int main() {
    int t;
    if (scanf("%d", &t) != 1) return 0;
    for (int tc = 1; tc <= t; tc++) {
        long long n;
        if (scanf("%lld", &n) != 1) return 0;
        printf("Case %d: %lld\n", tc, mates(n));
    }
    return 0;
}
```

14.4.6 Line by line

MOD. The prime $7\,340\,033 = 7 \cdot 2^{20} + 1$. Primality matters for Equation (14.6).

power. Fast exponentiation: square the base at each step and multiply it into the result when the current bit of the exponent is 1. It needs about $\log_2 M \approx 23$ steps.

INV3. Computed once, at program start, from Equation (14.6). Computing it instead of typing the constant removes one place where a typo could hide.

mates. Returns 0 for $N < 3$, outside the polynomial's domain. Otherwise it reduces N modulo M and runs Horner. Each step multiplies by v , reduces, adds the next coefficient, and reduces again. For the negative coefficients -56 and -2292 we add **MOD** before the closing `%`, so the intermediate value stays in $[0, M)$; in C++ the remainder of a negative number is negative, and a negative value would silently corrupt the next multiplication. The last line divides by 3 by multiplying with **INV3**.

main. Reads the number of queries and prints **Case k:** and the value for each.

TRAP. The three modular sins: (1) a negative intermediate value, because $(h * v - 2292) \% \text{MOD}$ can be negative in C++; (2) multiplying two values that were not reduced first, which can overflow 64 bits for larger moduli; (3) dividing with $/$ instead of multiplying by an inverse. A fourth trap is specific to interpolation: forgetting that the polynomial holds only from $N = 3$. Evaluating Equation (14.3) at $N = 1$ gives $f(1) = 236$ instead of 0, which is wrong.

14.5 Testing It: the Formula Against the Chessboard

The brute force in **Listing 14.4** is the most important program of this chapter: every coefficient of Equation (14.4) comes from its output, so it must be correct beyond doubt. It numbers the squares $r \cdot N + c$, tries every white king, black king and unordered pair of rook squares, and implements the rules of Section 14.4.2 literally. The function `rookHits` walks from the rook towards the target and stops at the first occupied square. In `mated`, the black king's own square is never marked as occupied, which implements "the king does not block behind itself". When the king captures a rook on square `s`, only the other rook is tested, and the square of the captured rook is temporarily marked free.

Listing 14.4 `brute.cpp`: counting checkmates by trying every position.

```
// BATTLECRY brute force: place a white king, a black king and two identical
// white rooks on an N x N board in every possible way and count the positions
// where the kings are not adjacent and the black king is checkmated.
#include <stdio>
#include <stdlib>
using namespace std;

int N;

bool adjacent(int a, int b) { // squares are numbered r * N + c
    int dr = abs(a / N - b / N), dc = abs(a % N - b % N);
    return dr <= 1 && dc <= 1;
}

// Does a rook on square r attack square t? occ[] marks the blocking squares.
bool rookHits(int r, int t, const bool occ[]) {
    int rr = r / N, rc = r % N, tr = t / N, tc = t % N;
    if (r == t || (rr != tr && rc != tc)) return false;
    int sr = (tr > rr) - (tr < rr), sc = (tc > rc) - (tc < rc);
    for (int x = rr + sr, y = rc + sc; x != tr || y != tc; x += sr, y += sc)
        if (occ[x * N + y]) return false;
    return true;
}

bool mated(int wk, int bk, int r1, int r2) {
    static bool occ[400];
    for (int i = 0; i < N * N; i++) occ[i] = false;
    occ[wk] = occ[r1] = occ[r2] = true; // the black king never blocks
    if (!rookHits(r1, bk, occ) && !rookHits(r2, bk, occ)) return false;
    for (int dr = -1; dr <= 1; dr++)
        for (int dc = -1; dc <= 1; dc++) {
            int x = bk / N + dr, y = bk % N + dc;
            if ((dr == 0 && dc == 0) || x < 0 || y < 0 || x >= N || y >= N)
                continue;
            int s = x * N + y;
            if (adjacent(s, wk)) continue; // walking next to the white king
            bool safe = true;
            if (s != r1) { // r1 survives; is s attacked?
                occ[r2] = (s != r2);
                if (rookHits(r1, s, occ)) safe = false;
                occ[r2] = true;
            }
            if (s != r2) {
                occ[r1] = (s != r1);
                if (rookHits(r2, s, occ)) safe = false;
                occ[r1] = true;
            }
        }
}
```

Listing 14.4 (Continued) *brute.cpp: counting checkmates by trying every position.*

```

        }
        if (safe) return false;           // the king escapes to s
    }
    return true;
}

int main() {
    int t;
    if (scanf("%d", &t) != 1) return 0;
    for (int tc = 1; tc <= t; tc++) {
        if (scanf("%d", &N) != 1) return 0;
        long long cnt = 0;
        int S = N * N;
        for (int wk = 0; wk < S; wk++)
            for (int bk = 0; bk < S; bk++) {
                if (bk == wk || adjacent(wk, bk)) continue;
                for (int r1 = 0; r1 < S; r1++)
                    for (int r2 = r1 + 1; r2 < S; r2++) {
                        if (r1 == wk || r1 == bk || r2 == wk || r2 == bk) continue;
                        if (mated(wk, bk, r1, r2)) cnt++;
                    }
            }
        printf("Case %d: %lld\n", tc, cnt % 7340033);
    }
    return 0;
}

```

The brute force costs about $N^8/2$ position checks: for $N = 10$ that is $5 \cdot 10^7$ positions, a few seconds, which is fine for a program run once at home. Its rule code is short enough to check line by line against Section 14.4.2, and there is a second, independent sign that it counts correctly: its values for $N = 9$ and $N = 10$ lie exactly on the degree-5 polynomial fitted to $N = 3..8$, which a miscounting program would be very unlikely to achieve.

The generator (**Listing 14.5**) asks for a few random board sizes between 1 and 9, and the usual **stress.sh** compares the formula with the brute force:

Listing 14.5 *gen.py*: random small board sizes.

```

# Random BATTLECRY input: a few board sizes small enough for the brute force.
import random
import sys

rnd = random.Random(int(sys.argv[1]))
t = rnd.randint(1, 4)
print(t)
for _ in range(t):
    print(rnd.randint(1, 9))

```

All 500 tests passed.

Because only nine sizes are possible, 500 random tests check each size many times. That is not wasted: it also checks the **Case k**: numbering with several queries per input.

14.6 Contest-Day Tactics

- **Try the stethoscope early.** Whenever the input is a single number N and the answer is a count, compute the first 8–12 values by brute force and look at the difference table. If it becomes constant, you are done in minutes. If it does not, try differences of $f(N)/N$, or separate even and odd N (some answers are polynomials on each parity class, called *quasi-polynomials*).

- **Use exact arithmetic for interpolation.** Expand Equation (14.2) with fractions (Python's `fractions.Fraction`, or by hand with a common denominator). Floating point loses the exact coefficients.
- **Do not trust the first values blindly.** The polynomial may start at some $N_0 > 1$. Look where the difference table becomes regular, and keep extra points for validation, as we did with $N = 9$ and 10.
- **When the modulus is not prime,** an inverse of 3 exists only if 3 does not divide the modulus. Otherwise avoid division by evaluating $f(N)$ directly with fractions reduced before the modulus, or use the Lagrange formula over the integers.
- **Know the cost.** As with every $O(1)$ formula in this book's toolbox [5], the resulting solution answers each query in $O(1)$: five multiplications. No array, no precomputation, and no chance of a timeout.

14.6.1 When the answer is not a polynomial

The difference table does not always become constant, and that is also information.

- If the differences look like a *constant multiple of the previous row*, the sequence grows exponentially, and the answer is likely a *linear recurrence* such as $f(N) = 3f(N-1) - f(N-2)$. Small brute-force values again determine the recurrence (the Berlekamp–Massey algorithm finds the shortest one automatically), and matrix exponentiation then evaluates it for huge N in $O(\log N)$ steps.
- If the table becomes constant only on even N or only on odd N , split the sequence by parity and treat each half separately.
- If nothing becomes regular, the answer probably depends on N in a more subtle way (divisors, primes, digits), and the brute-force values are still useful: look them up in your head against familiar sequences, or search for a pattern in their prime factorizations.

In every case the first step is the same as in this chapter: an honest brute force that you trust, and a table of values that you look at before you think about formulas.

14.6.2 Common mistakes

A few mistakes appear again and again when contestants use this pipeline for the first time.

- **A brute force that is subtly wrong.** Every coefficient inherits its errors. For chess problems the usual culprits are blocking (does a piece block a line or not?), captures, and the moving king's own square. Check two or three positions by hand, as in Section 14.4.2, before trusting any table.
- **Too few values.** Interpolating through exactly $d + 1$ values always produces a polynomial, whether or not the sequence is one. Keep at least two values for validation.
- **Starting the table too early.** If the first values belong to boards on which some shapes do not fit, the table looks irregular. Drop the first values one at a time until the differences settle; here that means starting at $N = 3$.
- **Fractions and floating point.** Coefficients such as $56/3$ cannot be stored exactly in a `double`. Clear the denominators and use a modular inverse, as Listing 14.3 does.
- **Forgetting to reduce N .** If N can be larger than the modulus, compute `v = n % MOD` first; otherwise the first multiplication may overflow.

14.7 Chapter Summary

- Counts of placements with a fixed number of pieces and local rules are often polynomials in N from some N_0 on, because each shape of position contributes a product of linear factors.
- Finite differences (Equation (14.1)) detect the degree: row d constant means degree d , with constant $a \cdot d!$.
- Lagrange interpolation (Equation (14.2)) recovers the polynomial from $d + 1$ values; extra values validate it.
- Horner's rule (Equation (14.5)) evaluates it with one multiplication per coefficient, and Fermat's theorem (Equation (14.6)) gives the inverse needed for fractional coefficients.
- For BATTLECRY, $3f(N) = 12N^5 - 56N^4 + 66N^3 + 554N^2 - 2292N + 2424$ for $N \geq 3$, and $f(N) = 0$ for $N < 3$.

14.8 Exercises

1. The values of a sequence for $n = 1..5$ are 2,9,28,65,126. Build the difference table, determine the degree and the leading coefficient, and guess the formula.
2. Use Equation (14.2) to find $f(10)$ for the cubic with $f(1..4) = 1,5,14,30$ by hand.
3. Compute 3^{-1} modulo 7 with Equation (14.6), and check it.
4. Evaluate Equation (14.4) at $N = 3$ with Horner's rule, showing every intermediate value, and divide by 3.
5. Count by brute force the placements of two non-attacking rooks (identical) on an $N \times N$ board for $N = 1..6$, detect the degree, and find the formula.
6. Explain why `(h * v - 2292) % MOD` can produce a wrong final answer even though C++ computes the remainder "correctly".
7. Why is it not enough to interpolate through *exactly* $d + 1$ values when the degree d was guessed from the table?
8. The number of squares of all sizes on an $N \times N$ board is 1,5,14,30, ... for $N = 1,2,3,4$. Which polynomial is this, and what is its value for $N = 100$ modulo $10^9 + 7$?

14.9 Solutions to the Exercises

1. Differences: 7,19,37,61; then 12,18,24; then 6,6. Degree 3, leading coefficient $6/3! = 1$. The formula is $n^3 + 1$: 2,9,28,65,126 fit.
2. With $x_i = 1,2,3,4$ and $x = 10$ the four basis values are $\frac{(8)(7)(6)}{(-1)(-2)(-3)} = -56$, $\frac{(9)(7)(6)}{(1)(-1)(-2)} = 189$, $\frac{(9)(8)(6)}{(2)(1)(-1)} = -216$ and $\frac{(9)(8)(7)}{(3)(2)(1)} = 84$. Then $f(10) = 1 \cdot (-56) + 5 \cdot 189 + 14 \cdot (-216) + 30 \cdot 84 = -56 + 945 - 3024 + 2520 = 385$.
3. $3^{7-2} = 3^5 = 243 = 34 \cdot 7 + 5$, so $3^{-1} \equiv 5 \pmod{7}$. Check: $3 \cdot 5 = 15 = 2 \cdot 7 + 1$.
4. Start with 12. Then $12 \cdot 3 - 56 = -20$; $-20 \cdot 3 + 66 = 6$; $6 \cdot 3 + 554 = 572$; $572 \cdot 3 - 2292 = -576$; $-576 \cdot 3 + 2424 = 696$. And $696/3 = 232$, as the brute force says. (Modulo M , the negative values become $M - 20$ and $M - 576$, and the result is the same.)

5. Two identical rooks do not attack each other exactly when they are in different rows and different columns: choose two rows, two columns, and one of two matchings, $\binom{N}{2}^2 \cdot 2$. The values for $N = 1..6$ are 0,2,18,72,200,450; the fourth differences are the constant 12, so the degree is 4 and the leading coefficient is $12/4! = 1/2$, which agrees with $\frac{N^2(N-1)^2}{2}$.
6. In C++, % keeps the sign of the left operand, so if `h * v < 2292` the result is negative. The next step multiplies a negative number by `v`, and the last `return` may print a negative number or, after later additions, a value that is off by a multiple of `MOD` but outside $[0, M)$. Adding `MOD` before % keeps every value in range.
7. Through any $d + 1$ points there is *some* polynomial of degree at most d , so interpolation always “succeeds”, even when the true sequence is not a polynomial or has a higher degree. Only values that were not used in the interpolation can confirm the result. That is why we fitted $N = 3..8$ and checked $N = 9$ and 10.
8. It is the sum of squares $1^2 + 2^2 + \dots + N^2 = N(N + 1)(2N + 1)/6$, the same sequence as in **Listing 14.2**: there are $(N - k + 1)^2$ squares of side k . For $N = 100$ the value is 338 350, which is smaller than the modulus, so it is also the remainder, as the program printed.

References

- [1] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1994.
- [2] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.
- [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] Sphere Online Judge, “BATTLECRY - Battle Of The Bastards - Kings Vs Rooks.” [Online]. Available: <https://www.spoj.com/problems/BATTLECRY/>
- [5] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

Chapter 15

Maximum Flow with Dinic's Algorithm — FASTFLOW

This last chapter brings together much of what the book has built: graphs, breadth-first search, depth-first search, careful 64-bit arithmetic and the discipline of testing against a brute force. The topic is *maximum flow*: how much can be sent from a source to a sink through a network whose edges have limited capacities? You will learn the idea of residual graphs and augmenting paths from zero, see why the max-flow min-cut theorem makes the answer checkable, meet the simple Edmonds–Karp algorithm as a warm-up, and then write Dinic's algorithm with *level graphs* and *current arcs*, fast enough for the SPOJ problem FASTFLOW with 5000 nodes and 30 000 undirected edges. The chapter, and the book, ends with a practical training plan: your road to the national podium.

15.1 Where This Helps: Water for the Orchard

A village cooperative brings water from a spring on the hill to a reservoir next to a fruit and durian orchard. Between them lies a network of pipes and channels, each with a maximum rate in liters per second, joined at small tanks where the water can split and merge. How many liters per second can reach the reservoir at most? And if the cooperative has money to widen a few pipes, which ones actually limit the supply?

Replace water by data and pipes by network links, or by roads and trucks, or by teachers and classes, and the same question appears: the largest amount that can move from a *source* to a *sink* when every connection has a capacity. Many assignment problems reduce to it as well: matching students to projects, workers to shifts, or contestants to computers, where each “unit of flow” is one assignment. Maximum flow is one of the most reusable models in algorithmics [1], and the national olympiad and its training camps treat it as advanced but fair material.

15.2 The Technique from Zero

15.2.1 Flows, capacities and cuts

A *flow network* is a directed graph with a source s , a sink t and a capacity $c(u, v) \geq 0$ on each edge. A *flow* assigns a value $f(u, v)$ to each edge with two rules:

$$0 \leq f(u, v) \leq c(u, v), \quad \sum_u f(u, v) = \sum_w f(v, w) \text{ for every } v \neq s, t \quad (15.1)$$

The first rule of [Equation \(15.1\)](#) says no pipe carries more than its capacity; the second says that water is neither created nor lost at an inner node. The *value* of the flow is the net amount leaving s . An *undirected* edge of capacity c , as in FASTFLOW, can carry up to c in either direction.

An s - t *cut* splits the nodes into a set S that contains s and a set T that contains t . Its capacity is the sum of capacities of edges from S to T . Every unit of flow must cross from S to T somewhere, so

the value of any flow is at most the capacity of any cut. The remarkable fact is that the bound is always reached:

THEOREM (max-flow min-cut). In every flow network the maximum value of a flow equals the minimum capacity of an s - t cut [2], [1].

This theorem is not only beautiful; it is practical. For tiny graphs we can compute the minimum cut by trying every set S , and that gives an independent brute force for testing, which we use in Section 15.5.

15.2.2 Residual graphs and augmenting paths

How do we find a maximum flow? Start with zero flow and repeatedly push more along some path from s to t . The catch is that an early choice can block a better one later, so we must allow *undoing* flow. The residual graph does exactly that. For every edge (u, v) carrying flow f it keeps two residual edges:

$$r(u, v) = c(u, v) - f(u, v), \quad r(v, u) = r(v, u) + f(u, v) \quad (15.2)$$

The forward residual capacity is what the pipe can still take; the backward residual capacity is how much of the current flow could be sent back, that is, cancelled. An *augmenting path* is a path from s to t in which every residual capacity is positive. Pushing the minimum residual capacity along it increases the flow value and keeps Equation (15.1) true. Ford and Fulkerson proved that a flow is maximum exactly when no augmenting path exists [2].

In code, every edge is stored together with a *partner* edge in the opposite direction, and pushing d units subtracts d from the edge and adds d to its partner. For a directed edge the partner starts with capacity 0. For an undirected edge of capacity c , both directions start with c , and they serve as each other's partner: pushing d from u to v leaves $c - d$ from u to v and $c + d$ from v to u , which is exactly right, because the flow from u to v can be cancelled and then c more can be sent from v to u .

15.2.3 Edmonds–Karp: shortest augmenting paths

Choosing augmenting paths carelessly can take a very long time; with irrational capacities it may not even terminate. Edmonds and Karp showed that choosing a *shortest* augmenting path (fewest edges), found by BFS, always needs at most $O(VE)$ augmentations, for a total of $O(VE^2)$ time [3]. This is the warm-up of Section 15.3.

15.2.4 Dinic: many paths per BFS

Dinic's algorithm [4] makes the shortest-path idea much faster by doing a whole *phase* of augmentations per BFS.

1. **Level graph.** A BFS from s in the residual graph gives each node its distance $level[v]$. Keep only residual edges that go from level k to level $k + 1$. Every shortest augmenting path uses only these edges. If t is unreachable, stop: the flow is maximum.
2. **Blocking flow.** Using DFS, push flow along level-graph paths from s to t until no such path is left. The result is called a *blocking flow*.
3. **Repeat** from step 1.

Figure 15.1 shows the level graph of the FASTFLOW sample: node 1 is at level 0, nodes 2 and 3 at level 1, node 4 at level 2. The edge between 2 and 3 joins two nodes of the same level, so the first phase ignores it; the self-loop at node 2 never carries anything.

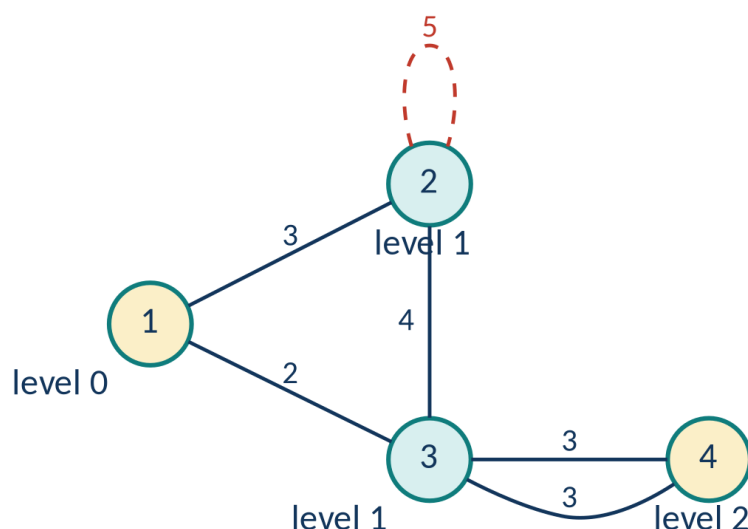


Figure 15.1 The FASTFLOW sample as an undirected graph with capacities and BFS levels. The dashed self-loop is discarded; the edge 2–3 lies inside level 1 and is not used in the first phase.

The engine of speed is the *current arc*. Each node v keeps an index $\text{arc}[v]$ into its edge list. During one phase, when an edge out of v turns out to be useless (full, or leading to a dead end), it stays useless for the rest of the phase, so the DFS never looks at it again: $\text{arc}[v]$ only moves forward. Each phase therefore examines each edge only a bounded number of times plus once per augmenting path, which gives $O(VE)$ per phase. Each phase increases the distance from s to t by at least one, so there are at most $V - 1$ phases, and the total is

$$O(V^2E) \quad (15.3)$$

On unit-capacity networks, such as bipartite matching, the bound improves to $O(E\sqrt{V})$ [5], and on real contest graphs Dinic is usually much faster than Equation (15.3) suggests. **Pseudocode 15.1** summarizes it.

Pseudocode 15.1 Dinic's algorithm with level graphs and current arcs.

```

MAX-FLOW( $s, t$ )
1  flow = 0
2  while BFS( $s, t$ ) reaches  $t$                                 // compute Level[]
3      arc[ $v$ ] = 0 for every  $v$ 
4      repeat
5           $d$  = DFS( $s, t$ , infinity)
6          flow = flow +  $d$ 
7      until  $d$  == 0
8  return flow

DFS( $v, t, f$ )
1  if  $v == t$ : return  $f$ 
2  while arc[ $v$ ] < number of edges of  $v$ 
3       $e$  = edge arc[ $v$ ] of  $v$ 
4      if  $e.\text{cap} > 0$  and level[ $e.\text{to}$ ] == level[ $v$ ] + 1
5           $d$  = DFS( $e.\text{to}, t, \min(f, e.\text{cap})$ )
6          if  $d > 0$ 
7               $e.\text{cap} -= d$ ; partner( $e$ ).cap +=  $d$ 
8              return  $d$ 
9      arc[ $v$ ] = arc[ $v$ ] + 1                                    //  $e$  is useless for this phase
10 return 0
    
```

15.2.5 Why each phase makes the sink farther away

Let δ be the distance from s to t at the start of a phase. After a blocking flow, every path of length δ in the residual graph has been destroyed: each one contained a saturated edge. Could new short paths appear? Pushing flow along an edge from level k to level $k + 1$ can create the reverse edge, which goes from level $k + 1$ back to level k , that is, *down* one level. A path from s to t must climb δ levels in total; every edge climbs at most one level, and every reverse edge goes down. So any path that uses a new reverse edge needs at least $\delta + 2$ edges, and any path of exactly δ edges would have to use only old level-graph edges, which the blocking flow has already blocked. Therefore the next BFS finds t at distance at least $\delta + 1$. Distances are at most $V - 1$, so there are at most $V - 1$ phases, which is where the factor V in [Equation \(15.3\)](#) comes from.

15.3 Warm-Up: Edmonds–Karp on a Matrix

For small graphs, an adjacency matrix of residual capacities makes the residual idea very concrete. **Listing 15.1** reads a directed graph, merges parallel edges by adding their capacities, and repeats: BFS for a shortest path, find the bottleneck, push it, and update both directions of every edge on the path as in [Equation \(15.2\)](#).

Listing 15.1 `edmonds_karp.cpp`: maximum flow by shortest augmenting paths on a capacity matrix.

```
// Warm-up: maximum flow in a small directed graph with Edmonds-Karp
// (always augment along a shortest path found by BFS), on a matrix.
#include <algorithm>
#include <cstdio>
#include <queue>
#include <vector>
using namespace std;

int main() {
    int n, m, s, t;
    if (scanf("%d %d %d %d", &n, &m, &s, &t) != 4) return 0;
    vector<vector<long long>> cap(n + 1, vector<long long>(n + 1, 0));
    for (int i = 0; i < m; i++) {
        int u, v;
        long long c;
        if (scanf("%d %d %lld", &u, &v, &c) != 3) return 0;
        cap[u][v] += c;           // parallel edges add up
    }
    long long flow = 0;
    while (true) {
        vector<int> parent(n + 1, -1);
        parent[s] = s;
        queue<int> q;
        q.push(s);
        while (!q.empty() && parent[t] < 0) {
            int x = q.front();
            q.pop();
            for (int y = 1; y <= n; y++)
                if (parent[y] < 0 && cap[x][y] > 0) parent[y] = x, q.push(y);
        }
        if (parent[t] < 0) break;           // no augmenting path is left
        long long push = -1;
        for (int y = t; y != s; y = parent[y])
            if (push < 0 || cap[parent[y]][y] < push) push = cap[parent[y]][y];
        for (int y = t; y != s; y = parent[y]) {
            cap[parent[y]][y] -= push;     // use forward capacity
            cap[y][parent[y]] += push;     // allow undoing it later
        }
        printf("augment by %lld\n", push);
        flow += push;
    }
    printf("max flow = %lld\n", flow);
    return 0;
}
```

On a small network with source 1 and sink 4, edges $1 \rightarrow 2$ (3), $1 \rightarrow 3$ (2), $2 \rightarrow 3$ (1), $2 \rightarrow 4$ (2) and $3 \rightarrow 4$ (3):

```
4 5 1 4
1 2 3
1 3 2
2 3 1
2 4 2
3 4 3
```

```
augment by 2
augment by 2
augment by 1
max flow = 5
```

The first two augmentations use the two paths of length 2, $1 \rightarrow 2 \rightarrow 4$ and $1 \rightarrow 3 \rightarrow 4$, each with bottleneck 2. The third uses $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ with bottleneck 1. The value 5 equals the capacity of the cut $\{1\}$ versus the rest, $3 + 2$, so by the max-flow min-cut theorem it is maximum. With V^2 memory and $O(V^2)$ per BFS this program is only suitable for a few hundred nodes, which is exactly why FASTFLOW needs something better.

15.4 Case Study: FASTFLOW

15.4.1 The problem in our own words

The SPOJ problem FASTFLOW [6] gives an undirected graph with $N \leq 5000$ nodes and $M \leq 30\,000$ edges. Each edge joins two nodes and has a capacity up to 10^9 ; the same pair may be joined by several edges, and an edge may join a node to itself. Find the maximum flow from node 1 to node N . As the name says, the solution must be fast.

15.4.2 Sample

```
4 6
1 2 3
2 3 4
3 1 2
2 2 5
3 4 3
4 3 3
```

5

By hand: the cut $\{1\}$ against the rest has capacity $3 + 2 = 5$ (the edges $1-2$ and $3-1$). The cut $\{1,2,3\}$ against $\{4\}$ has capacity $3 + 3 = 6$. No cut is smaller than 5, and Dinic finds the flow: in the first phase it pushes 2 along $1 \rightarrow 3 \rightarrow 4$ (the path through node 2 is a dead end, because 2 has no edge to level 2); the second BFS gives levels 1: 0, 2: 1, 3: 2, 4: 3, and the phase pushes 3 along $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, split over the two parallel edges between 3 and 4. After that node 1 has no residual edge left, and the total is 5.

15.4.3 Design decisions

- **Adjacency lists with partner indices.** Each `Edge` stores its head `to`, its residual `cap`, and `rev`, the index of its partner inside `adj[to]`. This gives $O(1)$ access to the partner and uses $O(V + E)$ memory.
- **Undirected edges as one pair.** One call `addEdge(u, v, c)` creates both directions with capacity c , which are each other's partner, as explained in Section 15.2.2.
- **Self-loops are dropped**, because a loop can never be part of a path from s to t . Parallel edges are kept as separate pairs; no merging is needed.
- **64-bit flow.** One edge carries up to 10^9 , and the total can reach $30\,000 \cdot 10^9 = 3 \cdot 10^{13}$. Every capacity, every pushed amount and the total are `long long`.

15.4.4 The complete solution

Listing 15.2 is the full program. The first half of the core, the BFS that builds levels, is outside the markers. The second half, the blocking-flow DFS with current arcs, lies between `// <P50>` and `// </P50>`.

Listing 15.2 `FASTFLOW.cpp`: Dinic's algorithm with level graphs and current arcs, $O(V^2E)$ in the worst case.

```
// FASTFLOW: maximum flow from node 1 to node N in an undirected graph with
// up to 5000 nodes and 30000 edges. Dinic's algorithm with current arcs.
#include <algorithm>
#include <cstdio>
#include <queue>
#include <vector>
using namespace std;

struct Edge {
    int to, rev;          // head of the edge; index of its partner in adj[to]
    long long cap;        // residual capacity
};

int n, m;
vector<vector<Edge>> adj;
vector<int> level, arc; // BFS level of each node; current arc of each node

void addEdge(int u, int v, long long c) {
    // undirected: each direction starts with the full capacity c, and the
    // two edges are each other's reverse in the residual graph
    adj[u].push_back({v, (int)adj[v].size(), c});
    adj[v].push_back({u, (int)adj[u].size() - 1, c});
}

bool bfs(int s, int t) { // levels in the residual graph; can we reach t?
    fill(level.begin(), level.end(), -1);
    queue<int> q;
    level[s] = 0;
    q.push(s);
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (const Edge& e : adj[v])
            if (e.cap > 0 && level[e.to] < 0) {
                level[e.to] = level[v] + 1;
                q.push(e.to);
            }
    }
    return level[t] >= 0;
}
```

Listing 15.2 (Continued) FASTFLOW.cpp: Dinic's algorithm with level graphs and current arcs, $O(V^2 E)$ in the worst case.

```
long long dfs(int v, int t, long long f) {
    if (v == t) return f;
    // <P50>
    // Try the edges of v from its current arc on; follow only residual edges
    // that go exactly one level deeper. Push along the first path that
    // reaches t, update both residual edges, and keep arc[v] where it stopped.
    // >>> YOUR TURN (Public Edition): write this part yourself, following the hint above.
    <<<
    // </P50>
}

long long maxFlow(int s, int t) {
    long long flow = 0;
    while (bfs(s, t)) { // one phase per BFS
        fill(arc.begin(), arc.end(), 0);
        while (long long d = dfs(s, t, (long long)4e18)) flow += d;
    }
    return flow;
}

int main() {
    if (scanf("%d %d", &n, &m) != 2) return 0;
    adj.assign(n + 1, {});
    level.assign(n + 1, -1);
    arc.assign(n + 1, 0);
    for (int i = 0; i < m; i++) {
        int u, v;
        long long c;
        if (scanf("%d %d %lld", &u, &v, &c) != 3) return 0;
        if (u != v) addEdge(u, v, c); // a self-loop carries nothing
    }
    printf("%lld\n", maxFlow(1, n));
    return 0;
}
```

15.4.5 Line by line

addEdge. The first `push_back` adds the edge $u \rightarrow v$; its partner will be the next element of `adj[v]`, whose index is the current size of `adj[v]`. The second `push_back` adds $v \rightarrow u$, whose partner is the edge just added, the last element of `adj[u]`. The order of the two lines matters: the index `adj[u].size() - 1` must be read after the first `push_back`.

bfs. A textbook BFS over edges with positive residual capacity. It returns whether the sink got a level.

dfs. The reference `int& i = arc[v]` makes the loop variable *be* the current arc: incrementing `i` moves `arc[v]` forward permanently for this phase. An edge is followed only if it has capacity left and goes exactly one level deeper. If the recursive call delivers `d > 0` units, the edge loses `d`, its partner gains `d`, and `d` is returned immediately. Note that on success the loop returns *before* `i++`, so the same edge will be tried again by the next DFS of the phase; that is correct, because it may still have capacity. On failure the loop advances past the edge for good.

maxFlow. For each successful BFS it resets all current arcs and calls `dfs` until it returns 0, adding every result to `flow`. The `while (long long d = ...)` form declares `d` in the condition and stops when it is 0.

main. Reads the graph, skips self-loops, and prints the maximum flow from 1 to N .

TRAP. (1) *Resetting the current arc inside the phase*, or not using it at all: the answer stays right, but the running time can explode and FASTFLOW times out. (2) *Modeling an undirected edge as two directed edges, each with its own zero-capacity partner*: correct but twice the edges; worse, forgetting one of them halves the capacity in one direction. (3) *int capacities or totals*: the answer can exceed 2^{31} . (4) Using `level[e.to] > level[v]` instead of `== level[v] + 1`: it still terminates, but it leaves the level graph and loses the guarantees of Section 15.2.4. (5) *Forgetting nodes numbered from 1*: arrays of size N instead of $N + 1$.

15.5 Testing It: Dinic Against the Minimum Cut

The max-flow min-cut theorem gives a brute force that shares no code with Dinic: try every set S of nodes with $1 \in S$ and $N \notin S$, add the capacities of the edges that cross, and take the minimum (**Listing 15.3**). For $N \leq 9$ there are at most $2^7 = 128$ such sets.

Listing 15.3 `brute.cpp`: the minimum cut by trying every set S .

```
// FASTFLOW reference for tiny graphs: by the max-flow min-cut theorem the
// answer is the cheapest cut, so try every set S with 1 in S and N not in S.
#include <cstdio>
#include <vector>
using namespace std;

int main() {
    int n, m;
    if (scanf("%d %d", &n, &m) != 2) return 0;
    vector<int> u(m), v(m);
    vector<long long> c(m);
    for (int i = 0; i < m; i++)
        if (scanf("%d %d %lld", &u[i], &v[i], &c[i]) != 3) return 0;
    long long best = -1;
    for (int mask = 0; mask < (1 << n); mask++) {
        bool in1 = mask & 1, inN = (mask >> (n - 1)) & 1;
        if (!in1 || inN) continue;
        long long cut = 0;
        for (int i = 0; i < m; i++) {
            bool a = (mask >> (u[i] - 1)) & 1, b = (mask >> (v[i] - 1)) & 1;
            if (a != b) cut += c[i]; // undirected edge crosses the cut
        }
        if (best < 0 || cut < best) best = cut;
    }
    printf("%lld\n", best < 0 ? 0 : best);
    return 0;
}
```

The generator (**Listing 15.4**) makes tiny multigraphs with loops, parallel edges, zero capacities, and in about a third of the cases capacities up to 10^9 to test 64-bit arithmetic.

Listing 15.4 `gen.py`: small random undirected multigraphs.

```
# Random FASTFLOW input: tiny undirected multigraphs with loops, parallel
# edges and occasionally huge capacities.
import random
import sys

rnd = random.Random(int(sys.argv[1]))
n = rnd.randint(2, 9)
m = rnd.randint(0, 20)
big = rnd.random() < 0.3
print(n, m)
for _ in range(m):
    c = rnd.randint(0, 10**9) if big else rnd.randint(0, 9)
    print(rnd.randint(1, n), rnd.randint(1, n), c)
```

All 500 tests passed.

For speed we generated two graphs at the maximum size: one with 30 000 random edges, and one layered graph with 49 layers of 100 nodes and six random edges from each node to the next layer, a shape that forces many phases. **Table 15.1** shows both finishing instantly.

Table 15.1 Running time of **FASTFLOW.cpp** on inputs of the maximum size.

Input ($N=5000, M=30\,000$)	Maximum flow	Time
random edges, capacities up to 10^9	3 904 368 904	0.01 s
49 layers of 100 nodes	49 995 537 200	0.01 s

Both answers exceed 2^{31} , the second by a factor of more than 20: a program with `int` flow would print garbage on exactly the kind of test a judge includes.

15.6 Contest-Day Tactics

- **Keep one Dinic in your head.** The code of **Listing 15.2** is about 60 lines. Practice typing it from memory until it takes under fifteen minutes and passes the brute force on the first try.
- **Model first, code second.** Most flow problems are *modeling* problems: the hard part is building the network [7]. Split a node into “in” and “out” joined by an edge of capacity k to limit how much passes through it; add a super-source and super-sink for many sources and sinks; use capacity 1 for “each item at most once”.
- **Read the minimum cut from the last BFS.** After the algorithm ends, the nodes the last BFS reached form the side S of a minimum cut. Problems that ask *which* edges to cut or *which* pipes to widen want exactly this set.
- **Check the edge count after modeling.** A model with 10^6 edges needs memory for $2 \cdot 10^6$ **Edge** records of 16 bytes: 32 MB. Estimate before coding.

15.6.1 A modeling example, worked

Three students, Adi, Bayu and Citra, must be assigned to three projects P1, P2 and P3, each project taking at most one student. Adi can do P1 or P2, Bayu only P1, Citra P2 or P3. Build the network: a source s with an edge of capacity 1 to each student, an edge of capacity 1 from each student to each project the student can do, and an edge of capacity 1 from each project to the sink t . A flow of value k corresponds to k assignments, because each student sends at most one unit and each project receives at most one.

A first augmenting path might assign Adi to P1. Then Bayu has nowhere to go: his only project is taken. The residual graph rescues us. The path $s \rightarrow \text{Bayu} \rightarrow \text{P1} \rightarrow \text{Adi} \rightarrow \text{P2} \rightarrow t$ uses the backward residual edge from P1 to Adi, which *cancels* Adi’s assignment to P1 and moves Adi to

P2. Finally Citra takes P3. The maximum flow is 3: Adi–P2, Bayu–P1, Citra–P3. This is exactly the undoing described by Equation (15.2), and it is why greedy assignment fails while flow succeeds.

15.7 Your Road to the National Podium

Fifteen chapters ago this book promised a road, not a miracle. A national medal, whether gold, silver or bronze, an honorable mention, or a place among the finalists, is the product of steady, well-aimed practice over many months. This section turns everything you have learned into a plan you can follow.

15.7.1 What the fifteen chapters gave you

Look back at the techniques as a toolbox: a contest-day playbook; segment trees for range queries, with and without updates; persistent segment trees for order statistics; monotonic stacks; state-space BFS and small-weight shortest paths; disjoint sets, minimum spanning trees and sweeps; Euler tours with Fenwick trees and binary lifting on living trees; dynamic programming for counting distinct objects and over bitmasks and profiles; CDQ divide and conquer; meet in the middle; suffix arrays; interpolation and modular arithmetic; and maximum flow. Above all, you have a *method*: understand the problem in your own words, find the model, design, estimate the cost before coding, implement cleanly, and prove the program right with a stress test.

15.7.2 A sixteen-week plan

Table 15.2 is a plan for the last four months before a national round. Adjust the dates to your calendar; keep the proportions.

Table 15.2 A sixteen-week training plan before a national round.

Weeks	Focus	Weekly routine
1–4	Foundations	10–15 easy and medium problems; retype one book solution per day from memory; stress-test every solution
5–8	Breadth	one chapter topic per week (trees, DP, strings, graphs); 3 problems per topic of rising difficulty; one past national problem
9–12	Depth	2 hard problems per week, each given at least 90 minutes before reading any hint; one timed virtual contest per week
13–15	Simulation	two full-length virtual contests per week under OSN conditions: same duration, no internet, only the allowed documentation
16	Taper	light practice, review of the error journal and of your templates; sleep and rest

15.7.3 Daily and weekly habits

- **One hard problem a day.** Choose it slightly above your level. Give it a real attempt before reading any solution; after reading one, close it and write the code yourself.
- **An error journal.** Every wrong answer, time limit or runtime error goes into a notebook with its cause: overflow, off-by-one, wrong order of calls, forgotten reset between test cases, misread statement. Reread it before every contest. The traps boxes of this book are the start of that journal.
- **Virtual contests.** Online judges such as the TOKI Learning Center (TLX) [8], Codeforces and AtCoder offer past contests you can take with a real timer. TLX also hosts past Indonesian olympiad problems, which are the best practice for the style and difficulty of OSN.

- **Templates in your fingers.** Fast input, a Fenwick tree, a segment tree, DSU, Dijkstra, the suffix array, modular power and Dinic should each take minutes to type and should be tested.
- **Upsolving.** After every contest, solve the problems you could not solve during it. Most of your improvement happens here.

15.7.4 On the day

- **Read every problem first.** Spend the first minutes reading all tasks and noting the subtasks. Points for partial solutions add up: a brute force that scores the small subtasks is often worth more than an unfinished full solution.
- **Follow the 45-minute rule.** If a problem has made no progress for 45 minutes, secure what partial score you can and move on; return later with fresh eyes.
- **Test before you submit.** Your own sample, the edge cases (smallest N , all equal values, the largest values), and, when time allows, a quick stress test like the ones in this book.
- **Stay calm when others seem faster.** You compete against the problems, not against the people around you.

KEY INSIGHT. A national title is the sum of small disciplines that never take a day off: one problem solved honestly, one error written down, one template tested. The podium is built from ordinary days.

Remember, too, that the road is long and that progress is rarely a straight line. There will be weeks in which nothing seems to improve, and then a week in which several ideas suddenly connect. Keep the routine through the flat weeks; they are when the foundations settle. Train with friends when you can: explaining a solution to someone else is one of the fastest ways to find the gaps in your own understanding, and a small study group keeps everyone going on the days when motivation is low.

15.7.5 A readiness checklist

Before the national round, check yourself honestly against **Table 15.3**. Each row describes something you can *show*, not something you *feel*.

Table 15.3 A readiness checklist for the weeks before a national round.

Area	Evidence that you are ready
Implementation	You can write Dinic, a segment tree and a suffix array from memory, and each compiles cleanly on the first attempt
Verification	You write a brute force and a generator for a new problem in under 15 minutes
Time management	In virtual contests you solve the easier problems quickly and use the 45-minute rule
Knowledge	You recognize the patterns of this book in unfamiliar statements
Stability	Your last four virtual contests all scored close to your best

15.8 Chapter Summary

- A flow obeys capacity and conservation ([Equation \(15.1\)](#)); its maximum equals the minimum cut capacity.
- Residual capacities ([Equation \(15.2\)](#)) let later augmentations undo earlier choices; an undirected edge is one pair of edges, each starting at full capacity.

- Edmonds–Karp augments along shortest paths; Dinic augments a whole blocking flow per BFS in the level graph, and current arcs make each phase $O(VE)$, for $O(V^2E)$ in total (Equation (15.3)).
- For FASTFLOW: adjacency lists with partner indices, self-loops dropped, parallel edges kept, and `long long` everywhere flow travels.
- The minimum cut over all subsets is an independent brute force for stress testing.
- The road to the podium is a plan: sixteen weeks, one hard problem a day, an error journal, regular virtual contests and honest self-checks.

15.9 Exercises

1. Run Edmonds–Karp by hand on the FASTFLOW sample (treat each undirected edge as two directed edges of full capacity) and list the augmenting paths.
2. In the FASTFLOW sample, find a minimum cut and verify that its capacity is 5.
3. Why must the current arc stay at an edge after a successful push instead of moving past it?
4. Show how to find a maximum matching in a bipartite graph (students and projects, each student at most one project, each project at most one student) with a maximum flow.
5. After `maxFlow` returns, how do you print the nodes on the source side of a minimum cut?
6. A problem limits how much flow may pass *through* each node v to k_v . How do you model this?
7. Why does FASTFLOW not need a separate partner edge for each direction of an undirected edge, while a directed edge always gets a partner of capacity 0?

15.10 Solutions to the Exercises

1. Shortest paths first: $1 \rightarrow 3 \rightarrow 4$ with bottleneck $\min(2,3) = 2$. Then $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ with bottleneck $\min(3,4,1+3)$; after merging the two 3–4 edges into one capacity 6, of which 2 is used, the bottleneck is $\min(3,4,4) = 3$. Total 5, after which node 1 has no residual capacity.
2. $S = \{1\}$: edges 1–2 (3) and 1–3 (2) cross, total 5. The flow of 5 shows no cut can be smaller.
3. A successful push may leave capacity on the edge (the bottleneck may have been further along the path). If the arc moved past it, later DFS calls in the same phase would miss capacity that is still usable, and the phase would end before the flow is blocking. That is still correct overall (the next BFS would find it) but can cost extra phases.
4. Add a source joined to every student with capacity 1, an edge of capacity 1 from each student to each acceptable project, and an edge of capacity 1 from every project to a sink. The maximum flow equals the maximum matching size, and the edges from students to projects that carry flow form the matching.
5. Run the BFS of `bfs` once more on the last residual graph (or reuse the last `level` array, which was computed by the failing BFS). Nodes with `level[v] >= 0` are reachable from the source in the residual graph and form the set S of a minimum cut.
6. Split every node v into v_{in} and v_{out} with an edge $v_{\text{in}} \rightarrow v_{\text{out}}$ of capacity k_v . Every edge that entered v now enters v_{in} , and every edge that left v now leaves v_{out} .
7. A directed edge $u \rightarrow v$ needs a backward residual edge to allow cancelling flow; that backward edge starts at 0 because nothing can be cancelled yet. For an undirected edge the backward

direction is itself a real edge with capacity c , so the two directions can serve as each other's partner: pushing d from u to v lowers $u \rightarrow v$ to $c - d$ and raises $v \rightarrow u$ to $c + d$, which is exactly the capacity of the real edge plus the cancellable flow. One pair of edges instead of two halves the memory and the work of the DFS.

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] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Canadian Journal of Mathematics*, vol. 8, pp. 399–404, 1956.
- [3] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *Journal of the ACM*, vol. 19, no. 2, pp. 248–264, 1972.
- [4] E. A. Dinic, "Algorithm for solution of a problem of maximum flow in networks with power estimation," *Soviet Mathematics Doklady*, vol. 11, pp. 1277–1280, 1970.
- [5] S. Even and R. E. Tarjan, "Network flow and testing graph connectivity," *SIAM Journal on Computing*, vol. 4, no. 4, pp. 507–518, 1975.
- [6] Sphere Online Judge, "FASTFLOW - Fast Maximum Flow." [Online]. Available: <https://www.spoj.com/problems/FASTFLOW/>
- [7] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 2. Lulu Press, 2020.
- [8] Tim Olimpiade Komputer Indonesia, "TLX — TOKI Learning Center." [Online]. Available: <https://tlx.toki.id/>